AFRL-IF-RS-TR-2000-59
Final Technical Report
April 2000

ADAPTIVE RESOURCE MANAGEMENT FOR DEPLOYABLE HPC SYSTEMS

Honeywell Technology Center

Sponsored by
Defense Advanced Research Projects Agency
DARPA Order No. E317

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

AFRL-IF-RS-TR-2000-59 has been reviewed and is approved for publication.

APPROVED: [illegible]

WAYNE A. BOSCO
Project Manager

FOR THE DIRECTOR: [illegible]

NORTHRUP FOWLER, Technical Advisor
Information Technology Division
Information Directorate

ADAPTIVE RESOURCE MANAGEMENT FOR DEPLOYABLE HPC SYSTEMS

Rakesh Jha

Contractor:	Honeywell Technology Center
Contract Number:	F30602-96-C-0321
Effective Date of Contract:	28 August 1996
Contract Expiration Date:	28 February 2000
Program Code Number:	6.2
Short Title of Work:	Adaptive Resource Management For Deployable HPC Systems
Period of Work Covered:	Aug 96 – Feb 00
Principal Investigator:	Rakesh Jha
Phone:	(612) 951-7320
AFRL Project Engineer:	Wayne Boxco
Phone:	(315) 330-3578

This research was supported by the Defense Advanced Research Projects Agency of the Department of Defense and was monitored by Wayne Bosco, AFRL/IFTB, 525 Brooks Road, Rome, NY.

REPORT DOCUMENTATION PAGE	*Form Approved* *OMB No. 0704-0188*

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE APRIL 2000	3. REPORT TYPE AND DATES COVERED Final Aug 96 - Feb 00

4. TITLE AND SUBTITLE ADAPTIVE RESOURCE MANAGEMENT FOR DEPLOYABLE HPC SYSTEMS	5. FUNDING NUMBERS C - F30602-96-C-0321 PE - 62301E PR - E317 TA - 01 WU - 01
6. AUTHOR(S) Rakesh Jha	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Honeywell Technology Center 3660 Technology Drive Minneapolis MN 55418	8. PERFORMING ORGANIZATION REPORT NUMBER N/A
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Defense Advanced Research Projects Agency, 3701 North Fairfax Drive, Arlington VA 22203-1714 Air Force Research Laboratory/IFTB, 525 Brooks Road, Rome NY 13441-4505	10. SPONSORING/MONITORING AGENCY REPORT NUMBER AFRL-IF-RS-TR-2000-59
11. SUPPLEMENTARY NOTES Air Force Research Laboratory Project Engineer: Wayne Bosco/IFTB/(315) 330-3578	
12a. DISTRIBUTION AVAILABILITY STATEMENT APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.	12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

Project goals were to develop techniques of continual reallocation of resources to maintain application performance despite statically unpredictable change in resource demands. Research was targeted to multiple application systems executing on HPC (High Performance Computing) platforms. This project built on the results of a previous program, called Adaptive Resource Allocation (ARA). In ARA, Honeywell developed techniques for dynamic reallocation of resources to single parallel applications, structured as multi-pipelines, executing on a high-performance parallel machine. They extended ARA results to systems with multiple applications and multiple machines connected over a network. In October 1997 DARPA merged the technical effort on this project with the RTARM project funded under Quorum. This did not affect the core statement of work for ARM, but led to extension of its completion date. ARM focused on developing an approach based on adaption models, and addressed best-effort resource allocation in an environment with partitionable rather than shared resources. Parallel HPG platforms were de-emphasized in favor of general distributed computing platforms. Results from ARM are being integrated into RTARM. The layered architecture of ARM has given way to a hierarchical architecture characterized by uniformity across different levels. The MPI-based communication infrastructure in ARM has given way to a CORBA ORB infrastructure. While ARM implementation was targeted to Unix machine connected over Ethernet, the target platform for PTARM consists of Windows NT machines networked over ATM.

14. SUBJECT TERMS High Performance Computing, Metacomputing, Adaptive Resource Management (ARM), Adaptive Resource Allocation (ARA)	15. NUMBER OF PAGES 102
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UL

Table of Contents

List of Figures

1. Summary

The project was started in September 1996. Its goal was to develop techniques for continual reallocation of resources to maintain application performance despite statically unpredictable change in resource demands. It was targeted to multiple application systems executing on HPC (High Performance Computing) platforms. It was anticipated that such adaptive capability would be needed in military systems such as SC-21.

As planned for this project, we built on the results of a previous program, called Adaptive Resource Allocation (ARA). In ARA, we developed techniques for dynamic reallocation of resources to single parallel applications, structured as multi-pipelines, executing on a high-performance parallel machine. We extended ARA results to systems with multiple applications and multiple machines connected over a network.

In October 1997, with DARPA approval, we decided to merge the technical effort on this project with the RTARM project funded under Quorum. This did not affect the core statement of work for ARM, but led to a 6-month extension of its completion date from November 1998 to May 1999. ARM still focused on developing an approach based on adaptation models, and addressed best-effort resource allocation in an environment with partitionable rather than shared resources. However, parallel HPC platforms were de-emphasized in favor of general distributed computing platforms. Some of the work we had completed, in particular the software infrastructure for managing multiple MPI-based applications, became less relevant.

Results from ARM are being integrated into RTARM. The layered architecture of ARM has given way to a hierarchical architecture characterized by uniformity across different levels. The MPI-based communication infrastructure in ARM has given way to a CORBA ORB infrastructure. While ARM implementation was targeted to Unix machines connected over Ethernet, the target platform for RTARM consists of Windows NT machines networked over ATM.

The work was performed jointly by Honeywell Technology Center and Georgia Institute of Technology, under Honeywell direction. This report describes only the work performed under ARM; hence, it represents an intermediate snapshot of the larger merged research.

2. Report layout

The report contains the following sections. A brief description of each section is given below to establish context before details are presented. The list of sections follows the list of tasks in the statement of work.

- **Program Objective** – This section describes the general characteristics of the targeted applications and the overall problem that ARM addresses.

- **Adaptation Models** – This section describes those attributes of applications and the underlying resources that are needed by ARM. Four distinct models are described –

    1. *Application Execution Models* capture the manner in which applications consume resources.
    2. *Performance and Timing Models* capture the performance requirements of applications in a system.
    3. *Decision Models* contain information about run-time detection of significant transitions in performance.
    4. *Resource Allocation Models* determine how to allocate and reallocate resources across applications and within applications
    5. *Enactment Models* describe when and how a new allocation should be brought into effect, given the potential cost and perturbation of reallocation.

    The main motivation for separating information into these models was to support a flexible architecture for ARM with plug and play capability.
- **ARM Architecture**: This section describes the layered adaptation architecture we developed. Each layer engages in negotiation, service translation, real-time monitoring and adaptation.
- **Real-Time Instrumentation**: We present an overview of the existing real-time instrumentation system, SPI, and describe the changes we made to it for ARM.
- **ARM Run-Time System**: This section describes the main components of the run-time support for adaptive resource management, including the software infrastructure.
- **Demonstrations**: This section describes the applications we demonstrated to show proof of ARM concepts.

Finally, we have attached a set of papers that represent the work performed under this project or built upon it.

3. Program Objective

Future defense systems will likely be characterized by dynamic variability in the performance demands of their applications. Many embedded DoD applications will be reactive, as they must interact with changes in an external physical environment. Often their run-time behavior will also be heavily data-dependent, depending on scene parameters, sensor modality, range to target, etc. Consequently, their computing resource requirements will tend to vary considerably during execution, and for the most part be statically unpredictable.

We refer to such systems as deployable systems. Given their time-varying and irregular resource needs, it will be necessary to manage resources dynamically.

Without dynamic adaptation in resource allocation, either computing platforms for deployable systems will have to be oversized or they will fail to meet the application requirements. In addition, in future military systems, the demand for higher agility will further require applications to be adaptive.

Effective management of computing resources in such an environment, and the adaptation of individual application subsystems is a challenging task. Deployable systems are different from the computing systems used in ground-based command and control operations over geographical dispersed areas. In deployable systems, applications are often interdependent; the performance requirements are usually stringent, and the applications tend to be more dynamic because they are embedded in a potentially rapidly changing environment.

The objective of ARM was to provide adaptive resource management mechanisms for specific models of applications, computing environments and resource usage. Adaptation is viewed in terms of continual allocation and reallocation of resources among the applications constituting a system to meet system-wide objectives.

4. Adaptation Models

4.1 Application Models

An application model determines how resources are requested and consumed. Some applications may be distributed across multiple computers. For example, the front end of a sensor-based application may be implemented on a SIMD machine, whereas the back end object processing is often implemented on general purpose MIMD machines. We assume that the data-parallel components of applications are implemented on MIMD computers as SPMD programs, which is a common style for hand-written codes as well as codes produced by compilers for parallel languages such as HPF.

Multiple applications may run simultaneously on a computer. The nodes of a computer may be partitioned across applications using either space multiplexing or time multiplexing. In our research, we limited ourselves to space sharing as it much more common in commercial HPC computers. Time-sharing is beset with severe performance penalties due to context switching and the difficulty of co-scheduling an application's tasks on multiple nodes on multiple computers.

Workload Model

The workload is a simplified multi-pipeline where individual stages may be tagged as parallel programs. A pipeline is an acyclically connected set of stages. A stage has zero or more inputs and zero or more outputs. Stages are connected by connecting an output of one stage to an input of another. No inputs or outputs are unconnected. Signal sources are modeled as stages with no input, signal sinks as stages with no output. Currently, we assume that there is only one source and one sink.

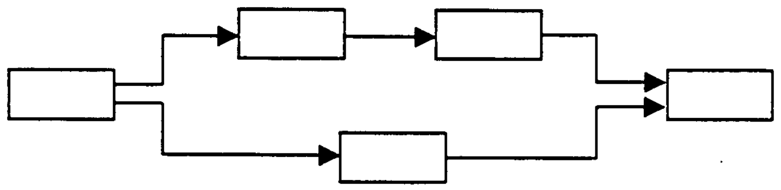


Figure 1: Multi-Pipeline

The intent is to use a recursive definition, so that even a connected subset of stages or just one stage, may be viewed as a pipeline. Invocation of a stage is input-driven. Output is always invocation-driven.

The more complicated the model is, the more complicated is the service request translation (SRT). We decided to stay with simpler models because SRT was not the central focus of our research.

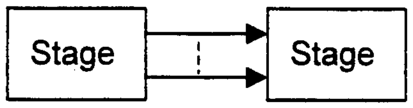


Figure 2: A pipeline is a recursive structure of stages

- The workload consists of a set of multi-pipeline applications, each with end-to-end QoS requirements.
- A multi-pipeline is a DAG of stages, each stage with zero or more in arcs (inputs), computational load, and zero or more out arcs (outputs).
- Stage invocation may be periodic (allowed only for source stages) or input-driven. An input-driven stage is invoked when a specified number of arrivals occur on each of its inputs.
- Computational load in a stage varies, depending on data sizes at input, and data-dependence.
- A stage issues output once after every invocation. The output data size may vary depending upon input data sizes.
- For parallel stages, a description of the parallelism in it.

Parallel programs are often described as task graphs (TG) consisting of tasks linked by communication edges. Task graphs have no temporal information about when the communications take place. We decided to use the Temporal Communication Graph (TCG) concept from Origami, which provides an unrolling of static task graphs in time. TCG and TG are static, in that the number of tasks and edges between them do not change at run-time.

Our target description expresses parallel computations as a function of the number of processors on which it is mapped. Our objective in a specification mechanism for temporal information was that we could provide a communication traffic description to NetEx (i.e. source, destination, and traffic pattern) when we transitioned from ARM to RTARM.

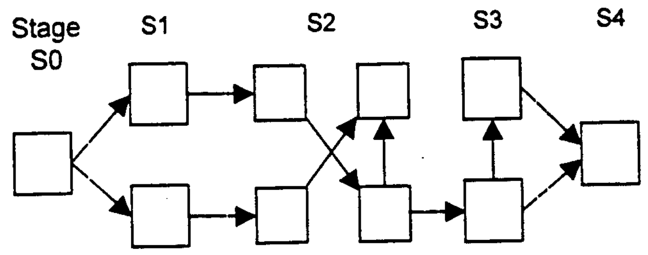


Figure 3: Workload Model: Pipeline with Parallel Stages

We use an enhanced version of TCG that makes it into a template for instantiation based on the number of processors allocated to it.

4.2 Performance and Timing Models

QoS is multi-dimensional, each dimension viewed as a range of values - low, high and a set of thresholds that define points at which some specific action is to be taken. For example, a drop in QoS below a threshold might trigger adaptation.

QoS includes quality dimensions and service dimensions. The quality dimensions include -

- Throughput as a function of input rate, reckoned at output.
- End-to-end latency between source and sink.

Service dimensions include per-stage specification of -

- Computational load as a function of input data sizes, and specification for each output data size as a function of input data sizes.
- Invocation rate

4.3 Decision Models

A critical component of the reallocation process is the decision model that determines when a reallocation of resources is necessary. As described earlier, applications are modeled as an acyclic graph of data-parallel tasks. Data frames are pipelined through this graph and each of these data-parallel tasks can be further structured as a collection of subtasks, each running on an individual processor. The number of subtasks within a task varies as processors are dynamically allocated to and deallocated from the original task. The subtasks are instrumented to provide performance measurements in real-time. Detectors process these instrumented streams of data to produce detection events, each signaling a major change in performance metrics. Decision models process these streams of detection events to determine if resource reallocation is necessary, and if so, to initiate procedures for the computation and enactment of new reallocations. In ARM, we address the reallocation of processors among tasks to maintain minimal frame latency through the task graph.

The majority of existing research on resource allocation and reallocation is focused on algorithms that determine how to most effectively allocate or reallocate resources. There is an extensive literature on dynamic resource allocation, typically in the context of load balancing algorithms. Strategies typically focus on where tasks must be scheduled as function of available resources. Research that is more recent has studied dynamic processor scheduling algorithms in multiprocessor systems and even algorithms for dynamic control of communication resources in parallel/distributed applications.

These resource allocation algorithms rely on the existence of a mechanism that determines when they are invoked, for example, at task arrival time. This does not permit reaction to run-time load variations within the application. We decided that for run-time reallocation, it is critical to be able to determine *when* such resource reallocation algorithms should be invoked during task execution. Accurate timing can avoid thrashing during transient workload changes, permit low latency reallocation, and in some instances preempt performance degradation by predicting reallocation needs.

Georgia Tech developed a combination of a low-latency decision model that is reactive in nature with a relatively more complex decision model that is predictive in nature. The model is quite insensitive to transient workload shifts or ``spikes", thereby reducing ineffective reallocations. The model is also quite effective in predicting impending workload changes. Thus, the decision model can be ``tuned" based on some knowledge of the application behavior. Using a synthetic benchmark generator, we experimentally demonstrated an increase in performance and a decrease in overhead across a range of input data parameters. While the current implementations are focused on a class of computationally intensive sensor-processing applications, these decision models are more generally applicable to asynchronous, event-driven computational models.

By coupling the reactive Bayesian model with the predictive Markovian model, we create a multi-level decision model capable of improving the performance of adaptive resource managers under a variety of input conditions. Under average input conditions, both models contribute to decrease the end-to-end latency of input frames and reduce the decision and enactment overhead. Toward the extremes, the Bayesian model proves more applicable to high noise environments and the Markovian model better suited to low noise environments. In these situations, the less suited model provides good backup support for the more effective model.

Under low noise conditions, the Bayesian level keeps track with the baseline model while the Markovian level pushed the system toward more acceptable performance states. Under high noise conditions, the Bayesian level filters a much larger percentage of the input spikes while the Markovian level ensured performance did not fall below the real-time specifications. Over a wide range of input streams, the coupled model is shown to maintain or improve the latency performance while decreasing the number of false triggers and unnecessary resource reallocations.

Ideas for future work include methods for dynamically varying the Bayesian and Markovian thresholds in response to the current task-level resource allocation, and implementing mechanisms for the Markovian model to suggest appropriate resource allocations for the predicted steady-state behaviors.

4.4 Resource Allocation Models

It is desirable that the underlying machines appear to the applications as one *virtual machine* that can be customized according to their individual and collective needs. This customizing should take place under control of the applications as well as automatically when a significant change in resource demands or availability is detected by the resource management system cognizant of applications characteristics.

4.4.1. Allocation and Assignment

Mapping an application to a heterogeneous target platform is a two-part problem:

- *Allocation*, which concerns the partitions of individual machines that are allocated to individual applications
- *Assignment*, which concerns the mapping of software components to specific processing resources, and may involve consideration of interprocessor communication behavior of the applications. We will use the term allocation (or mapping, configuration) to include both allocation and assignment from hereon.

The ARM system should provide continual on-line reallocation of resources to meet the overall mission objectives. The following types of events may trigger resource reallocation –

- Arrival and departure of applications
- Request by applications e.g. when an application knows it is about to enter a significantly different phase of computation
- Based on potential performance shortfalls detected by the ARM
- Request by the user, e.g. on a mode change

ARM can be effective only if the overhead of reallocation is significantly lower than the cost of doing no reallocation. Sufficiently fast algorithms are needed to compute a new allocation. Because of resource reallocation, application components may migrate across heterogeneous computers, with possibly significant change in the application's performance.

4.4.2. Resource Models

We adopted a hierarchical resource model, with a flat allocation model. For every resource in the system, a certain amount of resource is free, tested (for reservation), or reserved. For illustration, if a resource manager manages each resource, one can represent a parallel machine like the Cray T3D as a machine where the unit of allocation is the node.

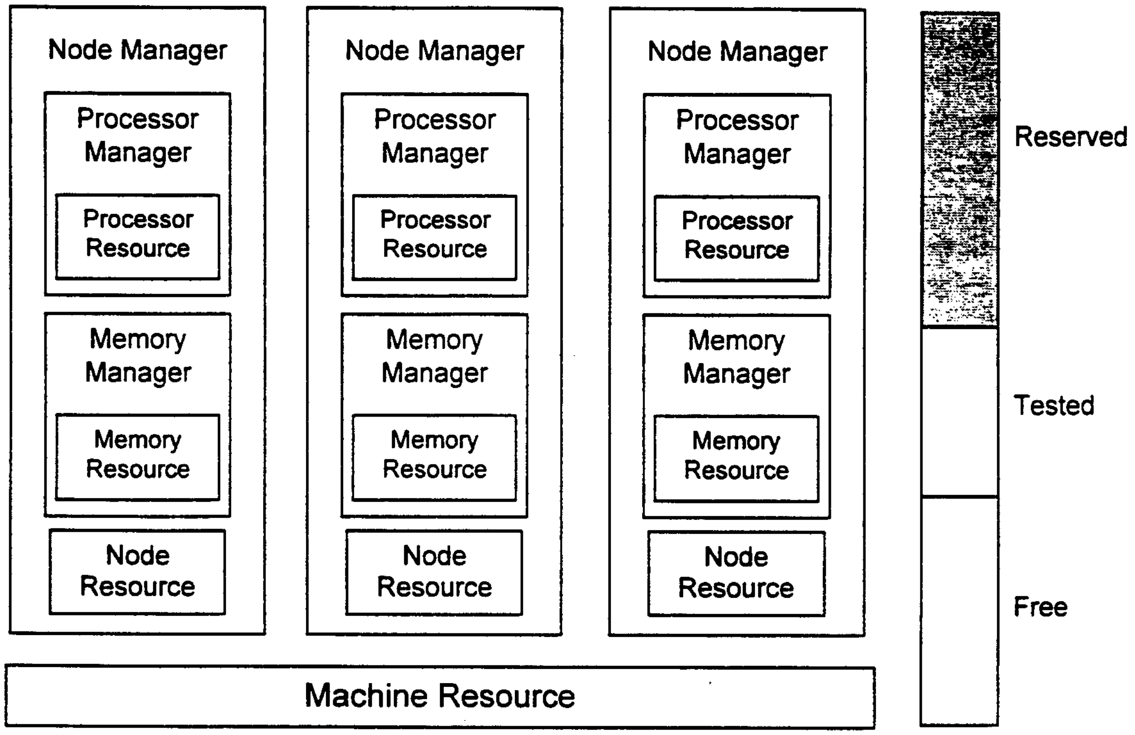


Figure 4: Illustration of Resource Model

Similarly, one can represent an IBM SP2 with several nodes, where the unit of allocation is a percentage of processor allocation.

5. ARM Architecture

ARM is divided into multiple layers, including an application layer and one or more resource layers. The application layer (A-Layer) is concerned with resource management issues relating to specific application models, and performance of the entire application rather than its parts. The resource layer (R-Layer) is common across many different application models, and encapsulates any hierarchy in the resources. For example, system layers in a network protocol stack belong in the R-Layer, and multiprocessor clusters may treat the cluster and individual multiprocessors as different layers. Potentially, there may also be a separate mission layer, which addresses mission-level objectives and tradeoffs across applications to achieve them. For now, only the A-layer and R-layer are considered.

Each layer is characterized by: workload model, QoS model, service requests, request translation and generation, negotiation and resource allocation, real-time monitoring, adaptation models and policies, and enactment. Note that the layered architecture described in this report has been generalized under RTARM to a hierarchical architecture.

The following sections describe the layers in more detail. It is currently assumed that the target applications are sensor-based multiple pipelines. Each layer receives a service request, translates it, and attempts to provide that service by negotiating for the services provided by lower layers. Existing already admitted requests might have to be squeezed through adaptation to release enough resources to admit new requests.

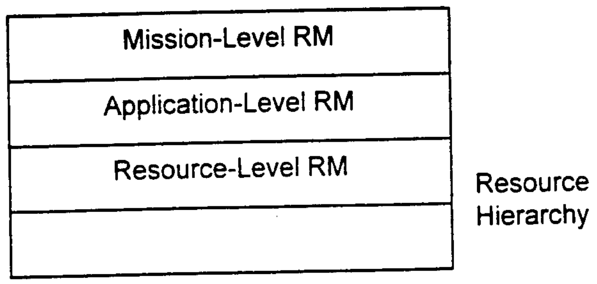


Figure 5: Layered Resource Management Architecture

Once a request is admitted and enacted, real-time monitoring allows the workloads and delivered QoS to be measured. Adaptations are triggered when the delivered QoS falls outside acceptable threshold regions. As described in the detailed sections, there is commonality among the possible adaptations. The Enactment component is responsible for bringing adaptations into effect.

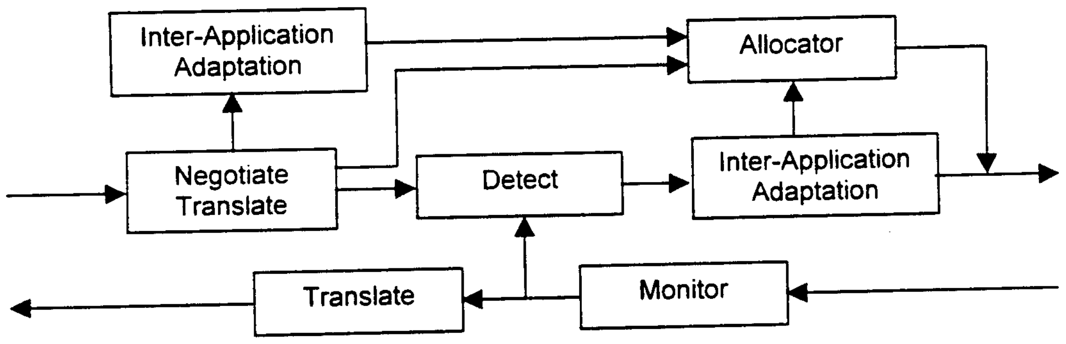


Figure 6: Resource Management Components in Every layer

Figure-6 shows the main components in each layer. The {R, A} arrows indicate the flow of service request and monitored or actual {QoS, Workload} respectively across layers.

5.1 Mission-Level RM

A mission is a set of applications, some of which may interact with one another. The set is dynamic as applications may arrive and depart dynamically. RM for applications is viewed in the context of a global mission-wide objective. Temporally, a mission may have several phases, with possibly different objectives and constituent applications. Transition between phases may be triggered by any of the general trigger conditions considered in ARM, i.e., operator action, detected failure to meet current objective, etc.

Services provided by the M-Layer

The M-layer manages the underlying resources in such a manner as to meet mission-level objectives. This service may be viewed as being provided to a mission (rather than to individual applications). QoS parameters associated with the service are chosen to represent the mission-level objectives. An example of a mission-level objective is to maximize the overall value of the application set.

5.2 Application-Level RM: A-Layer

Applications may have different programming- or computational models. For this effort, an application is a multiple pipeline, with possibly a reconfigurable structure, as described in Section 4. The A-layer does not understand missions, but manages resources for applications to meet their individual QoS requirements.

Services provided by the A-Layer

The A-layer provides resource management for individual applications or their components. It translates the incoming service request and QoS requirements and generates requests to the R-Layer are for computational services, memory, and network services. The requests may be made for each service separately, or jointly. For example, the request for computation and memory services may be made together if the A-Layer wishes to constrain the allocation to be co-located.

The A-layer also monitor application-level QoS of individual applications, which requires computing this QoS from information about the delivered QoS from the R-layer. Hence, the A-layer must at least monitor the actual values for all components in the application representation. Note that we are assuming composability of application-level QoS from its component-level QoS, which is a valid assumption for the multi-pipeline applications.

Adaptation triggers include QoS violation of entire application or substructures, explicit request from the M-Layer, and detection of failure in lower layers. The A-layer decides if application-level adaptation is needed. Possible adaptations are:

- Adjust the requested QoS of the application components in a way that does not violate application-level QoS delivered to the upper layer. Such adjustment may be localized to a subset of an application or it may be application-wide.
- Without changing requested QoS, use the services of the R-Layer to perform ARA-style redistribution of already allocated resources. This is based on transfer of resources from application components experiencing better than requested QoS, to components with worse than requested QoS.

5.3 Resource-Level RM: R-Layer

The R-layers represent resource hierarchies. In general, a platform consists of computers connected over networks. Each computer may be a Symmetric Multi-Processor (SMP), a distributed memory massively parallel machine (MPP), or a uniprocessor. Networks may include LAN's and high-performance interconnects providing shared memory.

A SMP consists of processors and memory shared among all processors. A distributed memory MPP consists of MPP-nodes connected by a MPP-network, where MPP-nodes consist of one or more processors and memory shared between them. A workstation consists of a processor and memory.

Service provided by the R-Layer

The R-Layer manages computing, network and memory resources for whole or subsets of multi-pipeline structured applications. The R-Layer does not understand applications, although can do application-wide resource management when the pipeline structure submitted to it is for an entire application. The QoS parameters in the request from the A-Layer are those associated with multi-pipeline application components (e.g. nodes, arcs) and structures.

Requests to the R-Layer are for computational services, memory, and network services. The requests may be made for each service separately, or jointly. For example, the request for computation and memory services may be made together if the A-Layer wishes to constrain the allocated resources to be co-located.

The R-layer translates incoming service request QoS parameters to QoS parameters for individual processors and links, for example in the case of MPP's. The R-Layer monitors the delivered performance and performs low-level adaptation. As in all layers, adaptation triggers include QoS violation, and explicit request from the A-Layer.

5.4 Architecture Evolution

As mentioned earlier, the layered architecture described in Section 4 has been generalized into a hierarchical architecture for resource management. As applications are built on top of services and services may be built on top of lower level services, resource management for the entire system is viewed as a hierarchy of service managers. Each node in the hierarchy can provide support for admission control, QoS translation, resource allocation, real-time monitoring, adaptation and enactment. The attached paper "Hierarchical architecture for real-time adaptive resource management" describes the generalized RTARM architecture.

6. Real-Time Instrumentation

We used the Honeywell Scalable Programmable Instrumentation (SPI) system for real-time monitoring. SPI offers the capability of monitoring a heterogeneous system in terms of traditional metrics such as latency and execution times, as well as metrics that depend on application semantics. Compared with other monitoring approaches, SPI allows the construction and evaluation of arbitrary detectors using predefined as well as user-defined actions and it also allows distributed coordination of all instrumentation activity and data.

Under this effort, we extended SPI in several ways - extensions to accommodate dynamically arriving and departing applications, and integration with the resource management system.

Georgia Tech used their Falcon system for real-time monitoring. Falcon can detect significant changes in a number of performance metrics. These monitors produce instrumented streams of sampled parameter values. Sample parameters include subtask execution time, subtask communication time, communication volume, input frame rates, and other measures of application performance or resource utilization. It is also possible to monitor application-specific measures such as the frequency of specific message types, access patterns to internal data structures or any other measure that is representative of the application's resource usage. Detectors operate on these streams to produce detection events corresponding to potentially significant deviations in performance guarantees.

7. ARM Run-Time System

This section describes implementation of the main components of the ARM run-time system, including the ARM Layers, Multi-Application Infrastructure, and the ARM Control Infrastructure.

7.1 Multi-Application Infrastructure

The core of this implementation is an infrastructure to control the processes of an MPI application dynamically by shrinking and expanding the number of processes in a graceful manner. We used the LAM version of MPI because of the dynamic process spawning capability that it provides to the user. This infrastructure contains a two-level resource manager system (system resource manager and application resource manager).

This infrastructure allows multiple applications to co-exist on the system under the control of a system resource manager. The system level resource management layer is between application level resource management and the operating system(s). The objective of the system resource manager is to continuously monitor and keep up the overall performance level as defined by the mission. This SW architecture is as shown in the following Figure. It enables:

- Applications to be spawned on multiple (distributed) processors
- Applications to receive a given QoS

  - Negotiation between the resource manager and the application
  - Dynamic reconfiguration of the number of application processes as and when the need arises
- Perform dynamic feedback adaptation operations within an application

ARM assumes that the application programs use MPI (not necessarily the LAM version). All application processes must call the ARM initialization procedure when they start and a termination procedure when they exit.

7.1.1. ARM Server

We implemented a central ARM server (system resource manager) and built utilities (and APIs) through which multiple application programs can execute in a controlled manner. Currently the server provides the following run-time services:

- Admit new applications
- Expand (grow) current application in size
- Shrink current application in size

A user or an application agent can request these services. At any point of time, the server maintains information about resources, applications and the binding of resources to applications. It also maintains two request queues: one for the currently active applications and one for newly admitted applications. These queues are maintained for only those requests that require new (additional) resources. For example, admit and expand both requires resources. In the case of shrink, the request is handled immediately. Four types of triggers invoke the scheduler.

- After an application admission
- After an application departure
- After an application shrinkage
- After an application expansion request

Currently, we have a simple FCFS scheduler that first considers the queue for the active applications, and then considers the queue for new applications for scheduling.

7.1.2. ARM Agent

ARM Agent is spawned automatically by the ARM server. Currently, for every application, the server spawns one agent, which in turn spawns the application processes. We also implemented synchronization protocols for information exchange among the ARM Server, ARM Agents, and the application processes.

Application Growth: The growth of an application (in terms of number of processes) can be initiated either by the server or by the agent. We have defined two types of protocols for their synchronization - a synchronous protocol and an asynchronous protocol. In the synchronous protocol, the server issues a "grow" command to the agent, which then informs all the application processes. If the agent gets back acknowledgement from all of the application processes within a certain time, it sends back an acknowledgement to the server. If this acknowledgement is received by the server before its timer expires, it send a commit message to the agent which then does the same to all of application processes and then the grow process takes place. In case the timer expires in either the server or the agent, they issue a cancel message to the appropriate parties immediately.

In the asynchronous protocol, the server is optimistic and sends only a single message to the agent and the agent is responsible to inform the server asynchronously about the success or failure of the operation. If the message does not reach the agent due to any reason, the server learns that only when the application departs.

Application Shrinkage: As with application growth, shrinking can be initiated either by the server or by the agent. If an agent is the initiator, it asynchronously informs the server, which then makes an update. If the server is the initiator, it goes through the protocols. For shrink we have implemented two protocols similar to those for application growth.

7.1.3. ARM Control Infrastructure

This implementation consists of the ARM layers for admission and adaptation control. This package consists of several integrated modules - admission control, real-time monitoring, and feedback adaptation. The ARM layers (A-Layer and the R-layers) are bundled as a single library package used by a centralized ARM controller for admitting new applications. The new applications request the service through an ARM server. The ARM controller was implemented as an Event-Action machine of the SPI (Scalable Programmable Instrumentation) system, which was extended to handle dynamic arrival of the processes to be monitored. The control software architecture is as shown in the following figure:

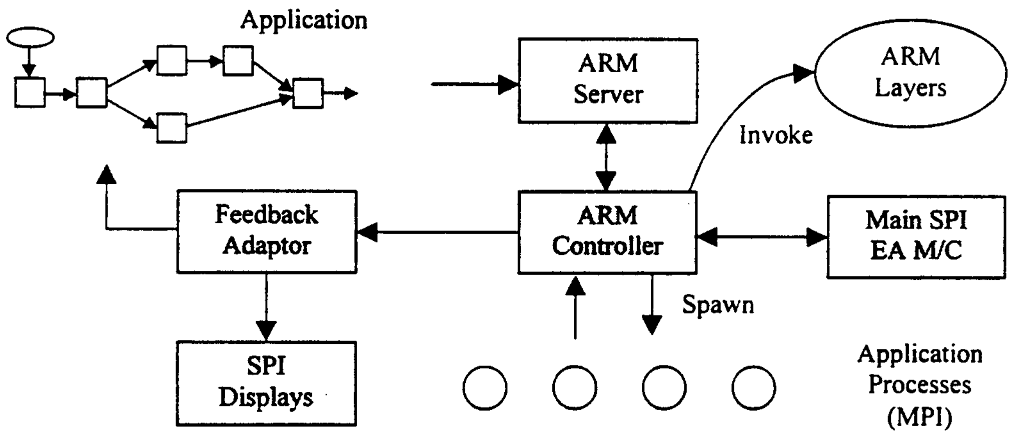


Figure 7: ARM Control Infrastructure

To start ARM, the LAM daemon is started by the user with the required hardware configuration. The user then starts the SPI loader, which starts the SPI main EA machine, the ARM controller, and the other required SPI EA machines. In the current implementation, there is a single ARM controller instance and multiple monitor instances. Each ARA monitor is associated with the set of application processes and a SPI real-time display.

To start an application, a user uses a client utility, which establishes connection with the ARM controller and forwards the user's request for admission, shrinkage or expansion. A new initialization protocol is added to all application processes to facilitate communication from the ARM controller to the application processes. This protocol requires application processes to establish a socket connection to the ARM controller. The application processes then continuously look for remapping messages from the ARM controller on this socket during execution.

7.2 ARM Layers

The control of a layer's functionality is embedded within a manager for that layer. These layer managers are responsible for allocation of resources and adaptation, using the services of the lower layer wherever necessary.

7.2.1. A-Layer

In the ARM implementation, the interface to the A-layer is through an object (class) called AppManager. The AppManager manager is a specialization of the Manager class. It contains objects such as the negotiator, allocator, enactor, detector and adaptor. The AppManager also has a reference to the R-layer manager (ResManager). This reference is created during the instantiation of the AppManager. The application (task) requests service from the A-layer using the method *TestAndHold()* of the AppManager object. The AppManager assigns an ID (task id), then uses its negotiator object to request appropriate service from the R-layer (since the A-layer by itself does not have resources).

The Negotiator translates the application request into one that is understood by the R-layer. This translation is called the forward translation and it involves translating task structures along with workloads and QoS. After translation, the negotiator makes a request to the R-layer manager through the reference maintained by the AppManager. Once the request returns, the negotiator translates back the assigned QoS into the one understood by the A-layer (backward translation). After this, the control passes back to the AppManager, which then reviews the returned QoS and returns it to the requesting task along with a task identifier. Further interaction between the requester and the AppManager takes place through the following methods using the assigned task id: *Reserve (), Release (), Abort ()*. Whenever resources are allocated, the AppManager maintains the task structures corresponding to the two layers along with their task ids and the resource allocation (QoS allocation) info in a hash table indexed by the task id. This is part of the *TestAndHold ()* method. The task model understood by the A-layer is the 'App' class, which inherits from both 'Task' class and the 'Graph' class.

7.2.2. R-Layer

The entry point for this layer is the ResManager object. It contains handles to the actual resources (managed by appropriate managers). The purpose of this object is strictly to provide a body for embedding the main control loop of the R-layer. The A-layer requests service from this object using the method TestAndHold (). The task model in R-layer is called the execution graph and is represented as a class named ExecGraph. The ResManager calls the negotiator of this layer to make request to the actual resource managers.

7.3 ARM Controller

The ARM controller is responsible for admission control and starting of application processes. Once applications processes are started, they send performance information to the ARM controller through SPI channels established during initialization phase of the application process. Depending on the application id (which is assigned by the ARM controller) of the process that is sending data, the performance data is routed to an appropriate monitor. Each application is assigned one monitor. When the ARA monitor decides to remap an application, it sends the new mapping to the ARM controller for that application. For this purpose, it uses a TCP/IP channel established between itself and all the application processes as part of the application initialization protocol.

8. Demonstrations

We developed three demonstrations for this project. The first demonstration was given in October 1997 on a network of Sun Solaris machines as shown in the Figure below. It showed QoS-based admission control and dynamic resource allocation for multiple synthetic sensor-based MPI applications.

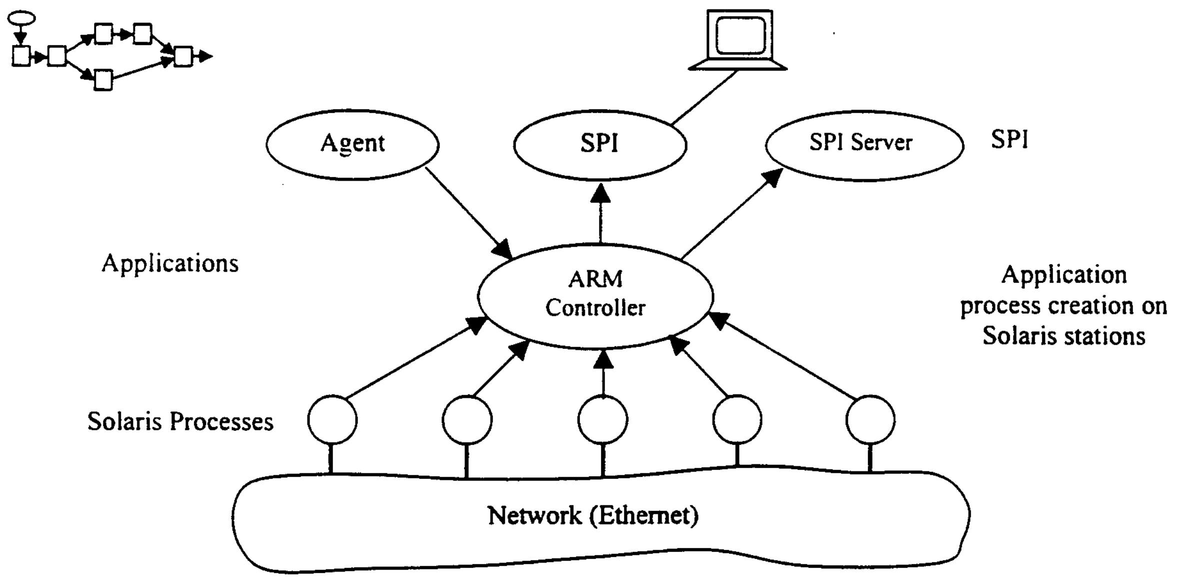


Figure 8: Vertical slice demonstration in October 1997

The vertical slice implementation included – a) a layered architecture for management of processor resources, b) admission control including QoS translation, and c) dynamic reconfiguration based on feedback of actual QoS (real-time monitoring, detection, and reallocation) within individual applications.

Georgia Tech contributed two demonstrations focusing on the utility of the technology and techniques developed in this project. They developed several additional applications with the objective of demonstrating specific levels of improvement.

Decision Models

Experiments using a synthetic workload generator and the statically defined decision model parameters yielded promising results. With the Bayesian decision model, we realized an overall reduction in unsuccessful invocations of the cost evaluator and number of unnecessary resource reallocations. This allowed more cycles for useful computation and masked the use of the more complex Markovian decision process. Experiments with frame latency showed similar or improved performance compared with the simple decision model for a significantly lower number of remappings.

Integration of the reactive Bayesian model with the predictive Markovian model improved latency and reduced false reallocations under a variety of input conditions. Under average input conditions, both models contributed to decreasing end-to-end latency and reducing the decision and enactment overhead. The Bayesian model proved better in high noise environments and the Markovian model proved better in low noise environments. In these situations, the less suited model provided good backup support for the more effective model. Under high noise conditions, the Bayesian level filtered a much larger percentage of input spikes while the Markovian level ensured that performance did not fall below the real-time specifications.

Vision Application

Georgia Tech evaluated some of their adaptation techniques on a vision application called Pfinder. The application consisted of a camera function, X-interface handler, and image processing functions. Adaptation was performed by reconfiguring the application mapping based on on-line monitoring of data flow rates.

Evolution of Demonstrations

Since the merger of this project with Real-Time Adaptive Resource Management (RTARM) in 1997, we targeted our demonstrations to the new hierarchical resource management architecture. A description of the technical features of the demonstrations is given in the attached papers.

9. Publications

The following publications based on this work are attached. Some of the publications describe research derived only partly from this project, and contain the results of subsequent continuing work.

- D. Ivan Rosu, K. Schwan, S. Yalamanchili, and R. Jha, "On adaptive resource allocation for complex real-time applications", in Proceedings of the 18th IEEE Real-Time Systems Symposium, San Francisco, December 1997.
- D. Paul, S. Yalamanchili, K. Schwan, and R.Jha, "Decision models for adaptive resource management in multiprocessor systems".
- M. Cardei, I. Cardei, R. Jha, and A. Pavan, "Hierarchical Feedback Adaptation For Real Time Sensor-based Distributed Applications"
- I. Cardei, R. Jha, M. Cardei, and A. Pavan, "Hierarchical Architecture For Real-Time Adaptive Resource Management".

On Adaptive Resource Allocation for Complex Real-Time Applications*

Daniela Ivan Roşu, Karsten Schwan,
Sudhakar Yalamanchili
Georgia Institute of Technology
801 Atlantic Drive, Atlanta, GA 30332-0208
{ daniela,schwan }@cc.gatech.edu
sudhakar.yalamanchili@ee.gatech.edu

Rakesh Jha
Honeywell Technology Center
3660 Technology Drive
Minneapolis, MN-55418
jha@src.honeywell.com

Abstract

Resource allocation for high-performance real-time applications is challenging due to the applications' data-dependent nature, the dynamic changes in their external environment, and the limited resources available of the embedded systems on which they run. These challenges may be met by use of Adaptive Resource Allocation (ARA) mechanisms that can promptly adjust resource allocation to changes in applications' resource needs, whenever there is a risk of failing to satisfy the application's timing constraints. Although not decided by the application, such adjustments satisfy the application's adaptation capabilities. ARA eliminates the need for 'over-sizing' real-time systems to meet worst-case application needs. This paper proposes an application model used to describe the application's resource needs and its adaptation capabilities. The model also describes the runtime variation of application needs. The paper also proposes a satisfiability-driven set of performance metrics for capturing the impact of ARA mechanisms on the performance of real-time applications. The relevance of the proposed metrics set is demonstrated experimentally, using an adaptive, synthetic application designed to represent time-critical applications in C^3I systems.

1. Introduction

Motivation. The resource management problems for real-time and embedded applications are exacerbated by the dynamic changes in their external environment and by the restrictions on resource availability. One commonly used solution is the worst-case resource allocation. In many cases this is not a realistic option because of the exceedingly high resource estimates resulted from complex interactions among the application components. If static resource allocation is not viable, adaptive methods must be used to adjust resource allocation to changes in the application's needs, therefore reducing the likelihood of failing to meet its real-time constraints.

*Funded in part by DARPA through the Honeywell Technology Center under Contract No. B09332478 and Contract No. B09333218, and by NSF equipment grants CDA-9501637, CDA-9422033 and ECS-9411846.

Contributions. This paper describes and evaluates models and mechanisms for *Adaptive Resource Allocation* (ARA) in the context of high performance, embedded applications. We consider applications with data-dependent execution, driven by event streams, composed by multiple, possibly parallel interacting components. Runtime changes in event rates and more importantly, in the data content of these events cause important changes in the resource needs of various application components. For such applications, it is simply not feasible to model accurately the per-event processing and communication needs. This class of applications includes radar systems [26], robots [7, 35, 39], target recognition, multi-object tracking, hypothesis testing [25].

ARA mechanisms can be used to promptly adjust resource allocation to changes in applications' resource needs, whenever there is a risk of failing to satisfy the application's timing constraints. Although not decided by the application, these adjustments satisfy its adaptation capabilities and eliminate the need for 'over-sizing' real-time systems to meet worst-case application needs.

This paper describes a novel model for capturing an application's adaptation capabilities by specifying the resource needs corresponding to each acceptable configuration. In addition, the model permits to capture the runtime variation of the resource needs caused by unexpected changes in application behavior.

Given the real-time nature of the applications targeted by this research, we propose to evaluate the ARA mechanisms by their impact on the satisfiability of the applications' real-time constraints. Specifically, we submit that it is essential to consider the latencies with which ARA mechanisms respond to changes in application needs when attempting to restore the satisfiability of real-time constraints. The quality of ARA decisions is evaluated with respect to how fast the application can return to acceptable performance and how good the performance in steady state is compared to the levels imposed by applications' real-time requirements.

In this study we identify elements that contributed to the effectiveness of ARA methods and heuristics. More specifically, we experimentally show the effects of early detection, enactment overhead, and incremental reallocation heuristics. **Assumptions and Experimental Environment.** In this work we assume that a multi-machine environment is destined to a single, complex application. As a result, performance perturbations are produced only by dynamics in the application's external environment or by changes in resource availability due to failures or explicit removals/additions. We also assume the explicit use of admission control mechanisms to guarantee sufficient resources to meet an application's initial required performance levels.

The models and heuristics proposed here are evaluated in the context of a centralized ARA controller. Online monitoring is performed with mechanisms described in [14]. Experiments are conducted with a synthetic application running on a cluster of workstations. The application is designed by Honeywell in the context of high performance C^3I[1] applications[25].

Related research. Previous work has described frameworks and mechanisms that facilitate the creation and use of online adaptation heuristics for real-time applications [5, 18, 22], including mechanisms for runtime monitoring, adaptation enactment, and mechanisms that ensure the reliable execution of applications [5, 22] or maintain high application throughput [18]. In comparison, the focus of this paper is not to define new frameworks, but instead, to define models and methods to be used in such frameworks and to analyze their effect on the adaptive applications.

Extensive research has addressed the problem of dynamic resource allocation for both the real-time [1, 3, 4, 9, 15, 17, 31, 40] and the non-real-time [13, 23, 27, 34] domains, typically considering dynamic resource allocation in the context of load balancing. However, the methods developed in these studies do not fit our target application model. This is because our model assumes that the resource needs of a time-constrained task, even when generated by the same type of event may vary throughout the execution of the application. This variability prevents us from using a periodic task model [15, 17] in which performance requirements are fixed throughout an application's execution, and therefore worst-case needs have to be considered. It also prevents us from using a sporadic task model, as in the real-time [9, 31, 40] or the non-real-time [13, 34, 23] domains, because of the high overhead of taking resource allocation actions at each task arrival. In addition, the specification of a real-time parallel task, as needed for an application component, is either too complex - in the real-time models, or incomplete - in the not-real-time models, because it does not describe the interaction among the parallel models of the same component.

[1]Command, Control, Communications and Intelligence

Resource reallocation triggered by runtime variation of application needs has received less attention. The schemes proposed for both real-time [4, 32, 17] and non-real-time [18, 23, 27, 38] domains do not consider the transitory effects of reallocation mechanisms on the satisfiability of application's performance constraints. In contrast, they are primarily interested in using adaptations to attain optimal average-case performance.

Overview of paper. In the remainder of this paper, we first identify the application and the ARA model driving our research (Section 2). In Section 3, we describe two important components of the application model used for ARA: the application resource usage model and the application adaptation model. In Section 4 we identify specific ARA performance criteria derived from the real-time nature of our target application. Last, in Section 5, we demonstrate by experiments the relevance of these criteria and identify methods that help improve ARA performance.

2. Real-Time Applications and ARA

Application Model. Our research targets reactive, high performance applications that must meet well-defined real-time constraints in dynamic execution environments. Each such application consists of multiple interacting *components* capable of executing in a distributed environment consisting of parallel machines, embedded-system components (e.g., signal processors), and user interface stations (e.g., workstations). Components are either sequential or parallel tasks and their resource needs may be data-dependent varying with changes in the rate or content of data inputs. In response, many components are programmed such that they can adapt their resource needs at runtime, by changes in their execution mode, algorithms or specific attributes such as the level of parallelism or communication protocols.

An application's execution is driven by *event streams* produced by the external environment or application components. Each event stream is processed by a fixed set of components, with fixed precedence constraints described by a *communication graph*. The input pattern of a stream may vary with changes in the execution environment. We use the term *intra-communication* to name the communication among parallel modules of the same application component, and the term *inter-communication* to name the communication between the component and its neighbors in the communication graph. We assume that, for each event, the intra-communication happens throughout the event processing while the inter-communication happens in a burst at the end of the source component computation.

The application's *performance requirements* are defined by constraints with respect to event rate, end-to-end latency, and inter-component relative completion delays. Each timing constraint may have specific bounds on its miss rate and/or burst.

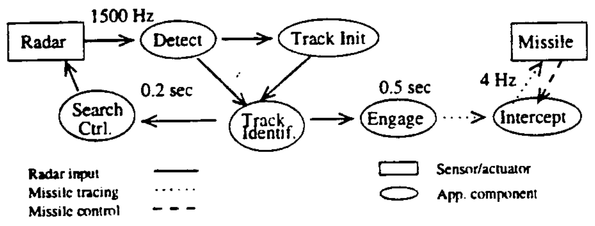


Figure 1. Radar Application

Sample Application. One sample application driving this research is a radar system. Figure 1 presents part of such a system, as described in [26]. *Detection*, *Track Init* and *Track Identif* are computation-intensive tasks, each well suited for parallel implementation [25]. Over time, their processing and communication needs vary with the number and characteristics (e.g., amplitude, direction) of dwells. Given the nature of their computation [25], these tasks can adapt by changes in their levels of parallelism.

The main event streams in the radar system are (1) the input from the radar, (2) the input from the missile tracking device, and (3) the missile control requirements. Timing constraints concern necessary event rates and processing latencies. For instance, the rate of the radar input is 1500Hz, and the missile control events must be sent at a rate of 4Hz. Additional constraints are: a 0.2 second-bound on the latency between *Detect*-ing a potential missile and engaging *Search Control*, and 0.5 seconds bound on the execution of *Engage*.

The radar system is one of the many applications concerned with processing signals from a sensor suite, forming hypothesis about and assessing the situation, and taking an appropriate response based on data observed and processed over a period of time. Other examples are multi-hypotheses tracking and image understanding[25]. Often the front end of these applications consist of signal processing stages whose computational needs are predictable, as they are independent of the signal values. However, computations at the back end depend on the semantic content of the signal values, being often heavily data-dependent.

Specific Resource Allocation Problems. The application model presented above poses interesting resource allocation problems. First, the event-stream-based execution makes viable the option of using long term resource allocation. Alternatively, a short term resource allocation based on dynamic real-time scheduling decisions [31, 3, 40], is prone to add a too much overhead to each event processing, in particular because the application components might often be parallel tasks executing in a distributed environment.

Second, the worst-case based allocation, the typical approach used in complex real-time systems, might not be appropriate for any application in our targeted class. In the context of data-dependent resource needs, it might be very difficult to evaluate the worst-case needs with enough accuracy to ensure both a safe execution and acceptable resource utilization. For example, in the radar system (see Figure 1), *Track Init* has very data-dependent needs as they vary with the number of dwell returns above a selected threshold and the ambiguity of spurious tracks. Thereby, the worst-case needs depended on the worst-case execution scenario, which makes them hard to evaluate and possibly very large compared to the needs of a typical execution scenario.

Our solution to these problems is to use adaptive resource allocation (ARA). By taking advantage of the application's adaptation capabilities, this method permits using long-term resource reservations while accommodating runtime changes in resource needs.

Adaptive Resource Allocation. ARA is a resource management paradigm that takes advantage of an application's ability of runtime adaptation in order to accommodate dynamic resource needs and to satisfy the system goals with respect to performance and resource utilization. In the context of our target application model, the goal of ARA is to insure that, at any time, the performance requirements of the application are satisfied.

In our approach, the ARA infrastructure can satisfy two types of resource requests: *explicit* and *implicit*. An explicit request is issued by the application upon a component arrival to the system, or whenever the application deems necessary to adjust its resource usage. An implicit request is issued by the ARA infrastructure itself, when changes in a component's resource needs considerably increase the likelihood of failing to satisfy of the application's performance requirements.

The implicit requests, and sometimes also the explicit ones, are satisfied by adjustments of the resource allocation of one or more application components decided by the ARA infrastructure itself. Such adjustments are called *automatic* because they are not explicitly required by the application. They are performed only when otherwise the performance constraints of the application are very likely to be violated, and they observe strictly the application/component specific adaptation capabilities. For example, an automatic adjustment might be performed when, due to the lack of resources in the system, a new application component can not be accommodated unless the allocation of other components is reduced. Similarly, an automatic adjustment can be triggered by an unexpected change in the execution environment that causes a change in the resource needs that can not be accommodated in the current configuration. For example, a change in the content of the input data may cause an increase of event processing time for a particular component that would require extending the component's level of parallelism in order to keep with the event rate.

In an alternative approach[5], the resource management infrastructure can satisfy only explicit requests, but it can provide the application with information on its observed resource usage. The resource usage adjustment decisions are made by the application itself.

In contrast, our automatic adjustments based approach permits to move part of the burden of the adaptation decisions from the application to the resource management infrastructure. A similar approach is taken in [18, 17] and, also, in our previous work [32]. The benefit of this approach is that unexpected changes in the application's resource needs are likely to receive faster response. Compared to the application, the resource management infrastructure has faster access to all the information related to the resource availability and current resource usage pattern of each application component. In addition, the application overhead with tracking the runtime variation of its requirements is eliminated. The drawback is that, compared to application-level decisions, the ARA decisions may fail to produce the most appropriate resource assignment for each particular situation. Likewise, ARA may result in changes in resource allocations not necessary for the good performance of the application. However, the models and mechanisms embedded in an ARA infrastructure can help minimize these drawbacks.

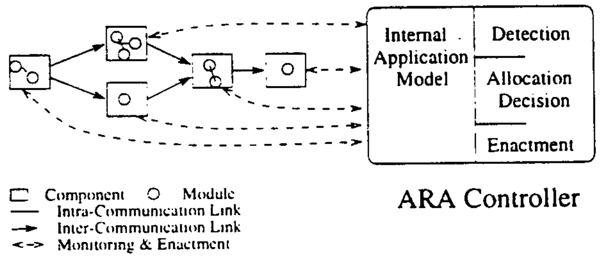


Figure 2. Centralized ARA controller

In order to achieve its functionality, the ARA infrastructure should include mechanisms for: (1) collecting information about application resource usage and resource availability; (2) detecting significant variations in application resource usage; (3) inferring the cause of observed variations and assessing the necessity of an automatic adjustment of the resource usage; (4) making decisions about resource assignments and automatic resource allocation adjustments; (5) notifying the application about significant changes in its resource usage; (6) notifying the application and the resource providers about changes in resource allocation and assisting them in the enactment of these changes. We assume that each application component capable of runtime adaptations has specific reconfiguration procedures that can be triggered by notifications of reallocation decisions received from the ARA infrastructure.

The ARA functionality is based on knowledge of the application characteristics. These characteristics are described by an *internal application model.* Besides the structure of the application (components, event streams, communication graphs) and its performance requirements, the model describes for each application component, the *acceptable configurations* (i.e., those instances of resource allocation that permit it to perform correctly) and the runtime variation of resource requirements. The model is used for the interpretation of monitored information, the estimation of system performance upon changes in resource allocation, and the guidance of decision heuristics. The internal application model is importantly influencing the way the ARA infrastructure can override the drawback with respect to the appropriateness of its decisions, and the execution overheads of the ARA mechanisms.

The performance of the overall ARA infrastructure and of each of its mechanisms reflects in the enabled application performance not only by how appropriate the resource allocation decisions are but also by how fast the ARA infrastructure responds to unexpected changes in application behavior. A short response time helps to reduce the intervals in which the application does not satisfy its timing constraints and to remain within the acceptable miss rate limits. Delayed ARA decisions or decisions that take too long to be enacted are less likely to reduce the risk of failing to satisfy the application's timing constraints.

In our work, the ARA functionality is provided by a module called *ARA controller.* This module can have a distributed or a centralized architecture. Figure 2 depicts a centralized controller, similar to the one used in our experiments. The controller's interaction with the application is restricted to monitoring and reallocation enactment.

In the next sections we will address the internal application model and the performance evaluation of an ARA infrastructure. Both these issues have significant impact on how the ARA can help an adaptive application to cope with unexpected changes in its resource usage and with restriction in resource availability.

3. Internal Application Model

This section describes the first novel contribution of our research. We propose models describing the application resource usage and its adaptation capabilities, both part of the internal application model maintained by the ARA infrastructure:

- The *resource usage model* (RUM) describes an application's expected the computational and communication needs and their runtime variation.

- The *adaptation model* (AM) describes an application's acceptable configurations in terms of expected resource needs and application-specific enacting overheads.

The RUM is used in the ARA decision making process to evaluate the current application's resource needs and to determine how the performance requirements will be satisfied. The AM permits the ARA controller to decide appropriate resource allocation adjustments without incurring any negotiation overhead as it is the case with other resource management solutions that support runtime adaptations[19]. In addition, the provision for estimations of the adaptation overheads permits the ARA controller to understand and evaluate tradeoffs between alternative adaptation strategies. In the remainder of this section we describe the two models.

3.1. The Resource Usage Model

Background. The *resources* available to the application are nodes and the communication links between them. A node is characterized by its speed (MIPS or MFLOPS) and the size of the local memory. Each node uses a scheduling policy able to guarantee the resource reservations and to provide feedback to the application on its actual resource usage, as those proposed in [20, 24]. A communication link provides a unidirectional connection between two nodes. It is characterized by one or more protocols (e.g., reliable, FIFO unreliable), with known available bandwidth and cost of I/O operations at each end-point - a constant per-message overhead and a per-byte overhead. For simplicity, the current RUM is based on uniprocessor nodes. Shared-memory multi-processors are modeled as sets of nodes, with equally distributed memory resources and connected by very high-speed communication links.

Model Formulation. The RUM describes the resource needs for each pair of application component and event stream. In the followings such a pair will called "a component".

Each component is described as an internally parallel task, with multiple cooperating modules that are independent from the point of view of resource allocation. The component's resource needs are described by two models - *static RUM* and *dynamic RUM*. The static RUM describes the expected computation and communication needs of the component, while the dynamic RUM captures the runtime variation of the component's needs with respect to the static RUM.

The parameters of the static RUM are the following:

- parallelism level;
- execution time;
- intra-communication protocol;
- intra-communication maximum message size sent;
- intra-communication total size sent;
- total number of intra-communication messages sent;
- inter-communication protocol;
- inter-communication size sent;
- total number of inter-communication messages sent;
- processor speed factor.

The inter-communication related parameters are defined separately for each component following in the event's communication graph.

The static RUM is specified by the application as part of an explicit request for resources. Its parameters can be estimated using traditional approaches like algorithm analysis or code profiling. The processor speed factor describes the performance of the node used for profiling.

Each parameter of the static RUM is assumed to be the largest value over the corresponding parameters of all the component's modules. This is equivalent to assuming that all modules have identical resource needs, the intra-component communication between any pair of modules is identical, and a module's incoming communication is the sum of all messages sent by all the other modules.

These assumptions keep the model safe and simple. However, the static RUM can be easily extended to describe the needs of each module of the application component. It can also be extended to include additional resource types as memory.

The *dynamic* RUM refers to those parameters of the static RUM that are likely to vary at runtime due to unexpected changes in input data content. The model is described by:

- execution factor;
- intra-component total size factor;
- intra-component maxim message size factor;
- inter-component total size factor.

Each factor represents the ratio between the maximum monitored performance of the corresponding metric over an application specific time interval and the static RUM specifications. The dynamic RUM is maintained by the ARA controller based on monitoring data received from the application.

Model Discussion. Given the static RUM, the ARA controller can obtain a good estimate of the component's computation and communication needs and use this information together with information on the event's input pattern and on the component deadline, to make per-resource schedulability analysis and reservations. The computation needs include the execution time and the computation related to performing the communication. The latter is estimated based on the number of I/O operations and the total amount transferred. The communication needs result direct from the model. Different from the typical real-time connection model [2], the static RUM does not model the intra-communication burst (the inter-communication being assumed bursty). This parameter is only related to the memory needs on the nodes and in the network. We ignore it because it can be substituted - for the node, by adding a memory parameter to the static RUM, and - for the network, by specifying a 'maximum message size' large enough to cover the maximum burst.

The dynamic RUM permits the ARA controller to make appropriate automatic adjustments even when the observed resource needs are larger than the application specifications. Such a situation may appear when the static RUM does not describe the worst-case needs, either because it was not possible to estimate them accurately or because the programmer decided so, possibly driven by the very small likelihood of situations where the needs get close to the worst-case limit.

The information needed to maintain the dynamic RUM could be obtained with low monitoring overhead from the instrumentation of the communication library.

Related Work. The resource usage model introduced here improves upon the deficiencies of real-time task models used in previous research [37, 10, 11, 16, 21] that do not permit for a low-complexity description of a parallel component. According to such models, a parallel application component should be described by a set of tasks with precedence constraints, each with fixed computation and communication needs, and with the I/O operations occurring only at the beginning and the end of a task (or event) execution. This would require each parallel component to be decomposed into multiple, small granularity tasks. If feasible, such complex decomposition would significantly increase the ARA decision overhead Despite its reduced level of detail that keeps the model simple and the decision overhead low, the RUM permits good estimates of the task performance.

The RUM also improves on previous parallel task models used in load balancing or task assignment problems [12, 17, 6, 27, 28, 29, 36] that do not describe the intra-communication needs. By considering these needs, the RUM enables a better resource management and a better estimation of the communication effects on the application performance.

3.2. Adaptation Model

Background. Each adaptive application component has several acceptable configurations. In general, the overhead of instantiating a new configuration has an application-independent and an application-dependent part. The application-independent overheads include the start-up of a new parallel module and the resource reservations (on the host and in the network). The application-dependent overheads, called here *adaptation overheads*, are determined by the component-specific reconfiguration procedures. We assume these are primarily determined by state transfers and initializations, and are significant when switching between configurations with different level of parallelism. We also assume that the ARA controller can evaluate the application-independent overheads.

Model Formulation. The adaptation model describes the acceptable configurations and the corresponding adaptation overheads for each pair of application component and event stream.

An acceptable configuration is described by: (1) *configuration id*, used by the ARA controller to notify the application about the changes in its resource allocation; (2) *static RUM*, specifies the resource needs as described in Section 3.1; (3) *adaptation overheads*, described separately for module start-up and shut-down. The adaptation overheads are described by: the amount of state to be transferred, and the execution time of the corresponding procedures (excluding communication).

The adaptation model is specified by the application upon an explicit request for resources. For each application component several acceptable configurations may be described. The ARA assumes that the static RUMs for all configurations in an adaptation model are compatible, in the sense of describing the requirements of solving the same problem in different configurations.

Model Discussion. The set of acceptable configurations permits automatic adjustments of a component resource usage without negotiation. The adaptation overhead permits the ARA infrastructure to estimate and control the enactment overheads, which can affect the short-term application performance.

Related Work. The inclusion of the adaptation overhead in the description of an acceptable configuration makes our model different from other schemes that allow the application to specify a set of acceptable configurations [1] at resource request time.

Our current model does not allow to specify the "value" each particular configuration brings to the application as in [1]. This is motivated by the current goal of our ARA: satisfy the application's performance requirements and with no concern for the overall "value" of the application. Anyway, our adaptation model can be easily extended to include a value parameter as well.

3.3. Using the Models

We briefly describe how the RUM and the adaptation model are used by the ARA infrastructure. Details can be found in [33].

The application requests an initial resource allocation by specifying an adaptation model. Based on the current resource availability, the ARA controller chooses an acceptable configuration, performs the corresponding reservations and notifies the application.

At runtime, each component is described by a *current RUM*. The static RUM corresponds to the acceptable configuration selected by the last allocation decision. The dynamic RUM is maintained based on the current static RUM and monitoring information.

When the likelihood of failing to satisfy the application's performance requirements increases above an application specific acceptable threshold, the ARA controller can decide an automatic adjustment of the application's resource allocation.

During the ARA decision, for each component, the static RUMs of its acceptable configuration are scaled by the corresponding dynamic RUM parameters. If for some component the current usage is larger than the current static RUM, the scaled static RUMs will describe needs larger than the initial specifications, prone to fit better the new application behavior. On the other hand, if the current needs of some component are lower than the specifications, the scaled static RUMs will describe smaller needs, enabling the ARA controller evaluate the unused resources and to take advantage of them in providing other components/applications with better service.

4. ARA Performance Characterization

The second contribution of our research is the proposal of a satisfiability-driven approach to evaluating the performance of an ARA infrastructure, different from the typical optimality-driven approach. In the context of a real-time application, we claim that the ARA infrastructure's reactivity is often more important than the optimality of its decisions. In addition, each ARA decision instance is equally important to the application, therefore we do not consider appropriate to measure the performance by averages over a large set of instances.

Our experiments show that delays in adjusting the resource allocation to changes in the application behavior increase the delays to reaching a safe steady state. Thereby, resource allocation characterized by large decision and enactment overheads, as an optimal decision is very likely to generate, increases the likelihood of failing to satisfy the application's timing constraints. For instance, in a heterogeneous distributed system, an optimal minimization of the end-to-end latency may require migrating all or many of the application components to more appropriate nodes. Such a reallocation decision may not be appropriate if during the enactment more events than acceptable miss their deadlines.

Focusing on the satisfiability of the application's performance requirements, we evaluate the performance of the ARA infrastructure by its response to a *single* variation in the application behavior that increases the risk of violating the performance requirements, called *critical variation*. Specifically, we consider the following metrics (see Figure 3):

- *reaction time* – the period between the occurrence of the critical variation and the completion of the correcting reallocation enactment;

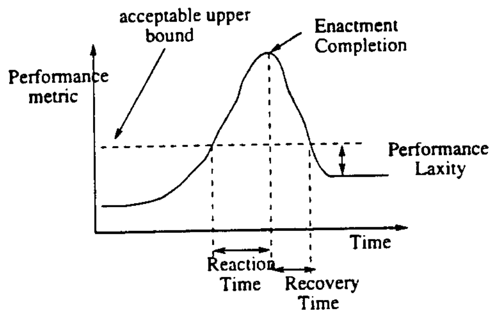


Figure 3. Performance Metrics for the evaluation of an automatic ARA decision

- *recovery time* – the interval between the enactment completion and the restoration of an acceptable performance level;
- *performance laxity* – the difference between the required performance, and the steady state performance after reallocation;

A good ARA controller is expected to have a low reaction time, low recovery time and large performance laxity.

These metrics reflect the effect of ARA mechanisms on the application's performance constraints satisfiability: recovery time and performance laxity relate to the quality of ARA reallocation decision, while reaction time relates to the overall ARA mechanisms: detection, decision, and enactment.

None of the above metrics can completely describe the ARA controller's performance. Specifically, performance laxity cannot measure the transitory effects of reallocation, while reaction time and recovery time do not reflect steady state improvement. Moreover, trade-offs exist between focusing on performance laxity vs. reaction time. Optimal performance laxity may result in reaction times that exceed acceptable delays due to high decision or enactment overheads.

When interested in characterizing the whole controller's performance, not only a single instance of critical variation, the reaction time and recovery time can be estimated by their maximums and the performance laxity by its minimum over all instances of critical variations.

The proposed metrics set is relevant for a real-time application. Poor reaction time and recovery time increase the time interval during which the application's performance constraints are not satisfied. Poor performance laxity increases the risk of failing to satisfy the constraints. Next section will demonstrate by experiments the relevance of reaction time.

Another interesting issue about the ARA infrastructure's performance is the *necessity* of automatic adjustments. The perturbation induced on the application by a not-necessary adjustment increases the risk failing to meet the constraints. Unfortunately, many times to assess the necessity of an adjustment requires knowledge on the future evolution of the system, which typically is not available. For instance a singular spike in CPU needs should not trigger an increase of the resource allocation for the corresponding component. We do not include this metric in our set, but we consider it when designing the mechanisms of ARA infrastructure. The necessity metric is related to detection and state assessment mechanisms (see Section 2).

Related Work. Previous studies considering automatic adjustments for ARA of real-time applications [17, 18] typically compare the performance attained with ARA against optimal solutions: [18] considers the performance loss with respect to an ideal ARA mechanism with instantaneous detection, optimal decision, and no overheads, while [17] focuses only on the optimality of the allocation decision. In contrast, we submit that the optimality of dynamic resource management is less important than the fact that an application's timing constraints are better satisfied.

5. Factors for ARA Reaction Time

In this section we consider the detection and the reallocation decision mechanisms and show how their design can affect the reactivity of the ARA controller, and consequently, the satisfiability of an application's performance requirements. Previous ARA related studies either considered application specific detection as a black-box[27], or performed detection mechanisms periodically at application independent intervals[17, 32]. Our experiments show that the ARA infrastructure performance is improved if the application characteristics and the current state are considered when choosing the methods for detection and allocation decision. In addition, our experiments show that the reaction time, component of satisfiability-driven metric set proposed in Section 4 is a relevant performance metric: the better the ARA infrastructure reaction time, the better application performance.

The experimental results reported in this study are obtained with a synthetic, distributed application designed by Honeywell in the context of high performance C^3I applications [25]. The application performs on a cluster of eleven UltraSPARC-I Model 170 workstations with an MPI-1 interface over 100Mbit switched Ethernet links. The application consists of multiple communicating components connected by an acyclic graph of communication links. Each component can adapt its execution to span over any number of processors. Each component module executes the following steps: (1) receive a message from each of the modules of the predecessor components, (2) execute according to the computation and intra-component communication pattern specific to its component, (3) send a message to each of the modules of the successor components.

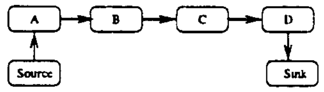


Figure 4. Configuration of Synthetic Application: 6-stage pipeline

In the following experiments the synthetic application has a pipeline configuration (see Figure 4). All events have the same type. They are periodically produced by the *Source*, consumed by the *Sink* and processed by the intermediate components. For each component, the step (2) mentioned above consists of: (2.1) exchanging a message with all of the modules in the same component; (2.2) computating for an amount of time that depends on: the parallelism level of the component and corresponding speedup coefficient; (2.3) exchanging messages as in (2.1). A stochastic model is used to emulate a step-like data-dependent variation of computation and communication needs.

Enactment is performed on event boundaries. The moment of performing the enactment (i.e., the id of the event before whose processing the resource exchange is performed) is determined by event currently processed by the closest predecessor of all of the components participating in the resource exchange. This method minimizes the enactment overhead because it requires no synchronization among donors, receivers and the components with which they communicate.

The adaptation overhead is small and identical for all components. In consequence, we do not consider it in reallocation decisions, but we do consider the application-independent resource reallocation overhead.

In the followings, "acceptable limit" for a particular performance metric is the upper bound derived from a corresponding performance constraint. In all the experiments the acceptable miss burst is one.

5.1. Detection

In this section we address the effect of early detection. The performance of a detection method is evaluate by: *promptness* – how soon after its occurrence, the critical variation is signaled; *trustworthiness* – what ratio of signaled variations is critical. The prompter the detector the earlier the detection is, and in consequence, the lower the ARA infrastructure's reaction time is. Detector trustworthiness is related to the necessity of the reallocation actions: the trustworthier the detector, the less risk to make a not-necessary adjustment. In the following experiments, any detection signal is triggering an automatic adjustment.

Promptness is more important than trustworthiness when the timing constraints are being violated. Figure 5

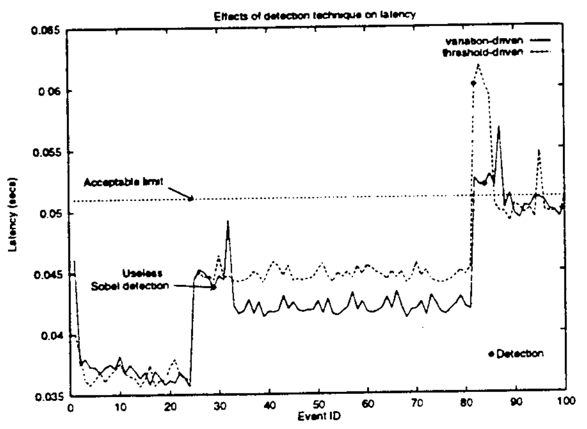


Figure 5. Promptness vs. trustworthiness: threshold-driven vs. variation-driven detector

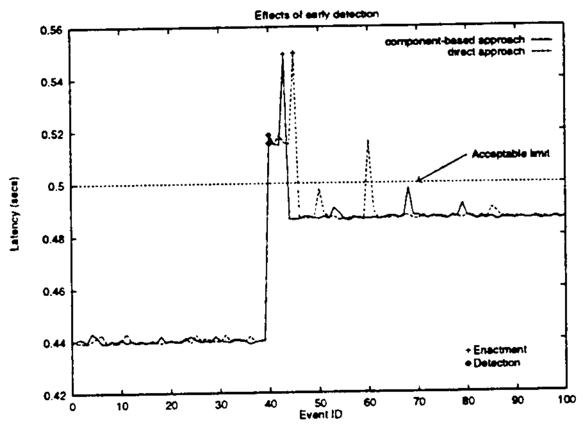


Figure 6. Effects on latency: component-based detector vs. direct detector

presents the influence of detection promptness on the end-to-end latency variation when the execution time of the bottleneck component is critical (i.e., very close to its events' inter-arrival intervals). We experiment with two detectors: a *threshold-driven* detector, which checks the sample value against the acceptable limit, and a *variation-driven* detector, which is similar to the Sobel detector used for edge-detection in computer vision [8, 30]. Among the two, the edge detector is prone to be trustworthier: it uses a range of points before and after the point if interest, uses smoothing techniques to eliminate the effects of noise. Unfortunately, these techniques result in a poor promptness. The threshold-driven is likely to be untrustworthy because it is sensitive to noise, but it is definitely prompt. The impact of a prompter detector is demonstrated in Figure 5 which shows (e.g., see Event ID 80) that the number of events failing the end-to-end latency constraint can be much larger with the variation-driven detector (smoothing size is 5, and sample-range size is 11).

On the other hand, a trustworthy detector can be used to detect changes in the application behavior which do not immediately cause the performance constraints to be violated, but which increase the risk of such a situation. In our experiment, a change in execution time which caused the end-to-end latency to get within 10% of the acceptable limit (see Event ID 25), is signaled by the variation-driven detector and triggers a reallocation which reduces the latency to more than 15% below the acceptable threshold. A threshold-driven can not be used for detecting changes that are not critical but increase the risk of failing to satisfy the performance constraints because of its sensitivity to spikes.

Promptness can be affected by method used to evaluate the metric of interest. Consider a performance metric that can be evaluated either directly or by composing several independent metrics. For instance, end-to-end latency can be measured directly or as it can be evaluated by the sum of execution and communication overheads of each application component on the event path. Consequently, for detection, one can take a *direct* approach by using the metric itself (e.g., the observed latency), or a *component-based* approach by using the component metrics (e.g., the observed execution of each application component on the event path).

The component-based approach is prompter than the direct approach and such improved performance results in shorter reaction times (see Figure 6, where component A's execution time increases). In particular, the difference is significant when the event path is long (in terms of latency) and the critical variation occurs early on the path.

5.2. Reallocation Decision

In this section we address the effects of considering enactment overheads and of using state-specific incremental heuristics for deciding automatic adaptations and corresponding resource allocations.

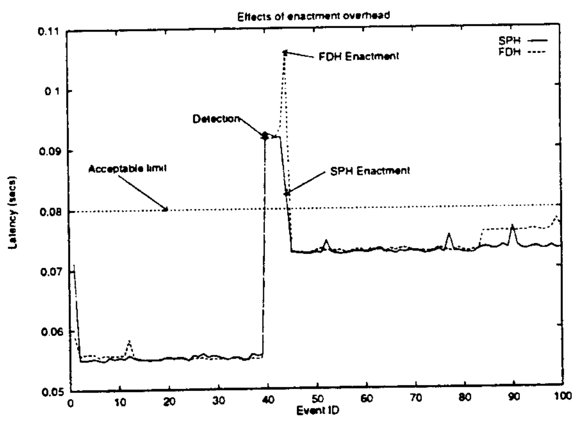


Figure 7. Influence of enactment overhead on reallocation results

Reallocation heuristics that are aware of the enactment overheads result in improved performance. Low enactment overhead improves reaction time, and reduces the risks of failing to meet application's timing constraints during the enactment period. Figure 7 shows the end-to-end latency variation with two decision heuristics distinct in their awareness of enactment overhead: First, the 'single-pair' heuristic (SPH) tries to accommodate a critical variation with a two-component transaction, which is likely to result in lower enactment overhead than transactions involving more components. Second, the 'fair-decrease' heuristic (FDH) tries to be fair about reducing the number of processors available to different application components but it ignores the enactment overhead.

To accommodate a step increase of component A's computation needs (see Figure 4), the SPH decides a 2-node transfer from component C to A, while the FDH decides a 1-node transfer from each of the components C and D to B. Both heuristics lead to similar steady state performance. However, the enactment overhead with FDH (23 msecs) is larger than with SPH (18 msecs). Thus, the number of events failing their latency requirements with FDH is larger.

Application-state driven incremental decisions can reduce the reaction time. An incremental decision can take advantage of the current system state in determining which components must receive or are allowed to donate resources. Such decisions usually provide more rapid response to performance perturbations than decisions which are computed using no history information [6, 17].

Simple incremental heuristics, such as determining a receiver and then searching for an appropriate donor, can give acceptable results with low decision overhead depending upon the order in which the components are checked. However, the effectiveness of an ordering criterion varies with the system state. We experiment with two ordering criteria: (1) by *actual execution time, AE,* and (2) by *execution time variation with reallocation, EV.*

In a rate-critical state, the primary goal of reallocation is to reduce the maximum execution time in the system. Thus, the bottleneck component needs to receive resources, and these resources can be taken from any other component, provided the resulting execution time does not violate the acceptable rate requirement. AE helps to focus immediately on the highest and lowest execution time components, while EV may search longer as it is very likely that the bottleneck will not realize the best improvement. In our experiment, AE order produces an acceptable reallocation after one try (1.34 msecs), while EV takes 4 tries (1.58 msecs). Note that in this experiment, a configuration analysis takes only 0.080 msecs. We expect this overhead to be larger for more complex application structures, when more complex timing requirements than end-to-end latency and maximum achievable event rate are considered.

In a latency-critical situation, the goal is to improve the sum of the execution times of all of the components on the critical path. The best solution with a two-pair transaction is to give resources to the component expected to have the largest reduction in execution time, and to take these resources from the component expected to have the lowest increase in execution time. By following this rule, the EV heuristic finds the best transaction after one try (2.56 msecs), while the AE takes 4 tries (2.83 msecs).

6. Contributions and Future Work

This paper considers the problem of ARA for high-performance real-time applications executing in dynamic environments. Applications consist of multiple parallel tasks with data-dependent resource needs. Our contributions are:

- present experimental results that demonstrate the importance of focusing on the response time of the resource allocation mechanisms rather than the optimality of their decisions, when real-time constraints must be satisfied.
- define an application resource usage model that permits to describe parallel real-time tasks and enables good reallocation decisions even when the observed performance is larger than the specified values.
- define an adaptation model that makes possible automatic ARA decisions and permits to evaluate the impact of enacting these decisions on the application's timing constraints.
- define a novel set of performance metrics to evaluate ARA performance by focusing on the satisfiability of the application's timing constraints. These metrics are reaction time, recovery time, performance laxity.
- identify factors related to detection and decision techniques which can influence the degree to which an application meets its real-time constraints. These factors are: early detection, enactment overhead, state-specific incremental decision heuristics.

The models and heuristics presented in this paper are shown useful in the context of processor reallocation for an adaptive, synthetic applications designed to represent time-critical applications in C^3I systems. In the future, we plan to apply them to other types of adaptive applications as a complex, distributed computer vision application. We also plan to integrate the insights and mechanisms presented here into a broader framework for resource management destined for systems where multiple real-time applications coexist, and where the ARA mechanisms described in this paper are used in conjunction with online negotiation mechanisms.

References

[1] T. F. Abdelzaher, E. M. Atkins, and K. G. Shin. QoS Negotiation in Real-Time Systems and Its Application to Automated Flight Control. *Real-Time Technology and Applications Symposium*, 1997.

[2] A. Banerjea and D. Ferrari. The Tenet Real-Time Protocol Suite: Design, Implementation, and Experiences. *IEEE/ACM Transactions on Networking vol.4, no.1*, Feb. 1996.

[3] A. Bestavros. Load Profiling in Distributed Real-Time Systems. *Journal of Information Sciences*, 1997.

[4] T. Bihari and K. Schwan. A Comparison of Four Adaptation Algorithms for Increasing the Reliability of Real-Time Software. *Real-Time Systems Symposium*, 1988.

[5] T. Bihari and K. Schwan. Dynamic Adaptation of Real-Time Software . *ACM Transactions on Computer Systems*, May 1991.

[6] S. H. Bokhari. Partitioning Problems in Parallel, Pipelined and Distributed Computing . *IEEE Transactions on Computers*, Jan. 1988.

[7] R. A. Brooks. A Robust Layered Control System For A Mobile Robot. *IEEE Journal of Robotics and Automations*, Jan. 1986.

[8] J. Canny. A Computational Approach to Edge Detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Nov. 1986.

[9] Y.-C. Chang and K. Shin. Optimal Load sharing in Distributed Real-Time Systems. *Journal of Parallel and Distributed Computing, pp.38-50*, 1993.

[10] S. Chatterjee and J. Strosnider. Distributed Pipeline Scheduling: End-to-End Analysis of Heterogeneous, Multi-Resource Real-Time Systems . *15th International Conference on Distributed Computing Systems*, 1995.

[11] S. Cheng, S. Hwang, and A. Agrawala. Schedulability-Oriented Replication of Periodic Tasks in Distributed Real-Time Systems. *15th International Conference on Distributed Computing Systems*, 1995.

[12] D. L. Eager, E. D. Lazowska, and J. Zahorjan. Speedup Versus Efficiency in Parallel Systems. *IEEE Transactions on Computers*, Mar. 1989.

[13] D. L. Eager, E. D. Lazowska, and J. Zahorjan. Adaptive Load Sharing in Homogeneous Distributed Systems. *IEEE Transactions on Software Engineering*, May 1986.

[14] G. Eisenhauer, B. Schroeder, K. Schwan, V. Martin, and J. Vetter. DataExchange: High Performance Communication in Distributed Laboratories. *9th International Conference on Parallel and Distributed Computing and Systems*, Oct. 1997.

[15] J. Huang and P.-J. Wan. On Supporting Mission-Critical Multimedia Applications. *3rd IEEE International Conference on Multimedia Computing and Systems*, 1996.

[16] K. Jeffay and D. Bennett. A rate-based execution abstraction for multimedia computing. *5th International Workshop on NOSDAV*, 1995.

[17] J. Jehuda. Automated Meta-Control for Adaptable Real-Time Software. *Real-Time Systems Journal*, (to appear).

[18] R. Jha, M. Muhammad, S. Yalamanchili, K. Schwan, and D. I. Rosu. Adaptive Resource Allocation for Embedded Parallel Applications. *3rd Int. Conference on High Performance Computing*, 1996.

[19] M. B. Jones, P. J. Leach, R. Draves, and J. I. Barrera. Modular Real-Time Resource Management in the Rialto Operating System. *5th Workshop on Hot Topics in Operating Systems*, pages 12–17, May, 1995.

[20] M. B. Jones, D. Rosu, and M. Rosu. CPU Reservations and Time Constraints: Efficient, Predictable Scheduling of Independent Activities. *"16th ACM Symposium on Operating Systems Principles"*, 1997.

[21] J. W. Liu, K.-J. Lin, and W.-K. Shih. Algorithms for Scheduling Imprecise Computations. *IEEE Computer Vol.24, No.5*, pages 58–68, May 1991.

[22] K. Marzullo and M. Wood. Making Real-Time Reactive Systems Reliable. *4th European SIGOPS Workshop*, 1990.

[23] C. McCann and J. Zahorjan. Processor Allocation Policies for Message-Passing Parallel Computers. *ACM Sigmetrics*, 1994.

[24] C. W. Mercer, S. Savage, and H. Tokuda. Processor Capacity Reserves for Multimedia Operating Systems. *"IEEE Int. Conference on Multimedia Computing and Systems"*, 1994.

[25] R. C. Metzger, B. VanVoorst, L. S. Pires, R. Jha, W. Au, M. Amin, D. A. Castanon, and V. Kumar. C3I Parallel Benchmark Suite - Introduction and Preliminary Results. *Supercomputing*, 1996.

[26] J. Molini, S. Maimon, and P. Watson. Real-Time System Scenarios. *Real-Time Systems Symposium*, 1990.

[27] D. M. Nicol and P. F. J. Reynolds. Optimal Dynamic Remapping of Data Parallel Computations. *IEEE Transactions on Computers*, Feb. 1990.

[28] K.-H. Park and L. W. Dowdy. Dynamic Partitioning of Multiprocessor Systems. *International Journal of Parallel Programming*, No.2, 1989.

[29] E. W. Parsons and K. C. Sevcik. Benefits of speedup knowledge in memory-constrained multiprocessor scheduling. *Performance Evaluation Review 27&28*, 1996.

[30] K. Pingle. Visual Perception by a Computer,Automatic Interpretation and Classification of Images. pages 277–284, 1969.

[31] K. Ramamritham and J. A. Stankovic. Dynamic Task Scheduling in Hard Real-Time Distributed Systems. *IEEE Software, Vol. 1, No. 3*, July 1984.

[32] D. I. Rosu and K. Schwan. Improving Protocol Performance by Dynamic Control of Communication Resources. *2nd IEEE ICECCS*, 1996.

[33] D. I. Rosu, K. Schwan, S. Yalamanchili, and R. Jha. On Adaptive Resource Allocation for Complex Real-Time Applications. *Georgia Inst. of Technology, GIT-CC-97-26*, 1997.

[34] H. Rotithor and S. Pyo. Decentralized Decision Making in Adaptive Task Sharing. *2nd IEEE Symposium on Parallel and Distributed Processing*, 1990.

[35] K. Schwan, T. Bihari, B. W. Weide, and G. Taulbee. High-Performance Operation System Primitives for Robotics and Real-Time Control Systems. *6th Symposium on Reliability in Distributed Software*, 1987.

[36] K. Sevcik. Characterization of Parallelism in Applications and Their Use in Scheduling. *Performance Evaluation Review, vol. 17*, May 1989.

[37] M. Spuri and J. A. Stankovic. How to Integrate Precedence Constraints and Shared Resources in Real-Time Scheduling. *IEEE Transactions on Computers, Vol. 43, No. 12*, pages 1407–1412, December 1994.

[38] A. Tucker and A. Gupta. Process Control and Scheduling Issues for Multiprogrammed Shared-Memory Multuprocessors. *12th ACM Symposium on Operating Systems Principles*, 1989.

[39] R. A. Volz, T. N. Mudge, and D. A. Gal. Using ADA as a programming Language for Robot-Based Manufacturing Cells. *IEEE Transactions on Systems*, Jun. 1984.

[40] H. Zhou, K. Schwan, and I. Akyildiz. Performance Effects of Information Sharing in a Distributed Multiprocessor Real-Time Scheduler. *Real-Time Systems Symposium*, 1992.

Decision Models for Adaptive Resource Management in Multiprocessor Systems*

Daniel Paul†, Sudhakar Yalamanchili†, Karsten Schwan† and Rakesh Jha‡

† Georgia Institute of Technology
Atlanta, GA 30332
{ dpaul, sudha }@ee.gatech.edu
schwan@cc.gatech.edu

‡ Honeywell Technology Center
3660 Technology Dr.
Minneapolis, MN
jha@htc.honeywell.com

Abstract

This paper addresses the problem of effective, on-line adaptive resource management in parallel/distributed architectures. The class of applications is very data-dependent, resulting in highly dynamic demands for available resources. Adaptive resource management is an alternative to engineering real-time systems toward worst-case application behaviors. To meet performance constraints, the system must react swiftly to run-time load variations and accurately redistribute resources in real-time. We propose decision models that operate on dynamically monitored performance data to determine *when* resource reallocation is necessary. The proposed decision models operate at two levels. The first is a low-level approach involving a Bayesian probabilistic decision model. The second is a high-level approach based upon state transitions and a Markovian decision model. The framework for our evaluation is a synthetic environment capable of simulating event driven, multitask applications where each task is partitioned into subtasks executing on individual processors.

1 Introduction

This research addresses a class of parallel applications that can be modeled as a collection of multiple, precedence-constrained data-parallel tasks or stages. We encounter such classes of event driven, data dependent applications that are very sensitive to run-time changes in event rates and input data content [4]. Consequently, execution is heavily data dependent and imposes highly dynamic resource demands upon the host system. A primary example of relevant applications are real-time defense systems that must constantly react to changes in an external physical environment. The environmental changes result in highly data-dependent processing loads. For example, Automatic Target Recognition (ATR) systems experience widely varying processing loads as a result of their heavy dependence on scene and algorithm parameters [1]. As the distances to targets of interest fluctuate, the number of regions of interest to process changes, resulting in significant variation of the computational

*Funded in part by DARPA through the Honeywell Technology Center under Contract No. B09332478 and Contract No. B09333218, and by NSF equipment grants CDA-9501637, CDA-9422033 and ECS-9411846 load. Because of their data-dependent nature, the resource requirements of the parallel tasks will vary significantly during run time.

A possible solution to meeting the performance requirements of such applications is to statically assign enough resources to accommodate the worst case application behavior. This solution is often infeasible because of the excessive level of resources required to address all possible situations[4]. The alternative is adaptive resource management to re-allocate limited resources dynamically in response to an application's needs. In real-time environments, efficient, low latency reallocation is crucial to the ability of such applications to meet deadlines. A critical component of this reallocation process is the decision model that determines when a reallocation of resources is necessary.

Our work is based on the operational model depicted in Figure 1. Applications are modeled as an acyclic graph of data-parallel tasks. Data frames are pipelined through this graph and each of these data-parallel tasks can be further structured as a collection of subtasks, each running on an individual processor. The number of subtasks within a task varies as processors are dynamically allocated to and deallocated from the original task. The subtasks are instrumented to provide performance measurements in real-time. These instrumented streams of data are processed by detectors that produce detection events signaling major changes in performance metrics. Decision models process these streams of detection events to determine if resource reallocation is necessary, and if so, to initiate procedures for the computation and enactment of new reallocations. In this paper we only address the reallocation of processors among tasks to maintain a minimal frame latency through the task graph.

The majority of existing research on resource allocation and reallocation is focused on algorithms that determine *how* to most effectively allocate or reallocate resources. There is an extensive literature on dynamic resource allocation, typically in the context of load balancing algorithms (for example see [8, 15, 21, 12, 18, 9]). Strategies typically focus on *where* tasks must be scheduled as function of available resources. More recent research has studied dynamic processor scheduling algorithms in multiprocessor systems[14, 13] and even algorithms for dynamic control of communication resources[16] in parallel/distributed applications. These resource allocation algorithms rely on the existence of a mechanism that determines when they are invoked, for example, at task arrival time. This does not permit reaction to run-time load variations within the application. We argue that for run-time reallocation, it is critical to be able to determine *when* such resource reallocation algorithms must be invoked during task execution. Accurate timing can avoid thrashing during transient workload changes, permit low latency reallocation, and in some instances preempt performance degradation by predicting reallocation needs. This focus on decision models complements (and is distinguished from) the recent work on the online adaptation of systems for real-time applications[18, 19]. Such frameworks incorporate mechanisms for run-time monitoring, adaptation enactment, and processor reallocation. We argue that effective decisions models must be incorporated into such frameworks if they are to be successfully applied to online adaptive resource management functions.

This paper proposes the effective use of decision models for dynamic resource allocation in high-performance parallel/distributed systems. Specifically this paper proposes a combination of a low latency decision model that is reactive in nature with a (relatively) more complex decision model that is predictive in nature. We show that such a model is quite insensitive to transient workload shifts or "spikes", thereby reducing ineffective reallocations. The model is also quit effective in predicting impending workload changes. Experiments are presented that relate characteristic of the application, such as noise, and parameters of the decision model. Thus, the decision model can be "tuned" based on some knowledge of the application behavior. Using a synthetic benchmark generator, we experimentally demonstrate an increase in performance and a decrease in overhead across a range of input data parameters. While the current implementations are focused on a class of computationally intensive sensor-processing applications, these decision models are more generally applicable to asynchronous, event-driven computational models. Throughout this paper the presentation will rely on an automatic target recognition (ATR) application to illustrate the behavior of the proposed model.

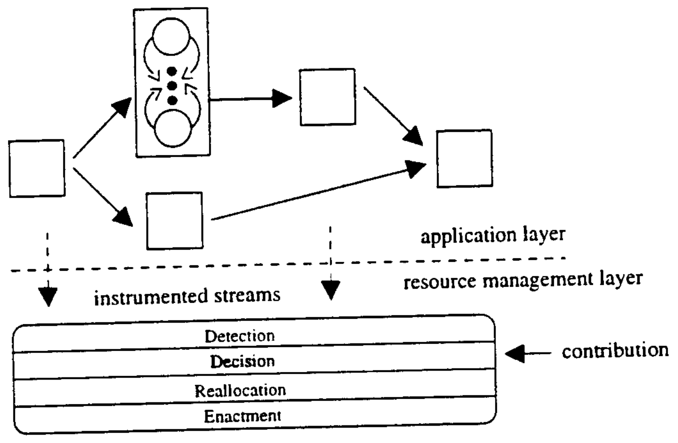


Figure 1: Operational model for dynamic resource allocation

The remainder of this paper is organized as follows. Section 2 provides a description of our system architecture and the problem we are addressing. In section 3, we provide a detailed description of our proposed decision models. Lastly, section 4 presents our experimental results and a review of the contributions of this paper.

2 Problem Description

2.1 System Overview

We consider an ATR application processing a stream of sensor data frames. The vision processing must extract targets from the background terrain and maintain track and identification information. The goal is to maintain a certain processing frame rate. As illustrated in Figure 1, resource reallocation is managed by a system consisting of four major components: detection, decision, reallocation, and enactment.

Currently, monitoring is accomplished by a real-time instrumentation system that can detect significant changes in a number of performance metrics [5] [6]. These monitors produce instrumented streams of sampled parameter values. Sample parameters include subtask execution time, subtask communication time, communication volume, input frame rates, and other measures of application performance or resource utilization. We may also choose to monitor application-specific measures such as the frequency of specific message types, access patterns to internal data structures or any other measure that is representative of the application's resource usage. Detectors operate on these streams to produce detection events corresponding to potentially significant deviations in performance guarantees. Decision models analyze streams of detection events to make assertions about the current global state and implicitly about the future state of the system. If a decision to reallocate is made, a cost evaluator creates a new specification of resource assignments. In this paper we only consider the (re)assignment of processors to tasks. Tasks are data parallel and the number of subtasks of a task is equal to the number of processors assigned to that the task. This new resource assignment must then be enacted and adopted by the system. In this case processors within one task may be reallocated to another task to maintain the frame rate.

2.2 Problem Definition

Monitoring and detection occur constantly as data frames move through the task pipelines. Decision models must constantly react to the detection event streams being produced by the detectors. However, the cost evaluation and resource reallocation only occur if the decision model signals the reallocation module. There is some computational overhead involved in calculating a new resource mapping, so ideally one would only incur this penalty under the assurance of improved overall system performance. Because of imperfect monitors and noisy input, it is difficult to determine accurately the optimal times to initiate a remapping. An effective reallocation decision policy must weigh the costs of remapping against the potential performance benefits [3]. In a worst-case scenario, a reallocation may be enacted only to discover that the previous conditions were transient and the system is more unbalanced than before remapping.

There are two competing factors governing the reallocation decision process. The first is the desire for fast detection and reaction. This is important because of the real-time requirements of the majority of these applications. Long and complex decision algorithms can result in large decision and enactment overheads. This overhead can cause multiple events to miss their deadlines because of quickly changing environmental conditions. In addition, many of the dynamic input conditions are highly transient and unstable. For example, background foliage that appears only in a few successive image frames can produce sudden, transient change in the processing workload. Therefore, the second important factor is the global performance of the application using the new resource mapping. Ideally, new resource mappings will lead to configurations which are stable and improve the long-term performance of the application. While quick decisions may result in locally improved performance, they are by nature not globally cognizant, i.e., will the change in terrain features persist for a relatively long period of time? Thus, it is possible to continually make locally optimal task-based decisions while the overall performance of the application steadily moves toward a less efficient state.

2.3 Solution Strategy/Approach

This paper proposes the use of a decision model structured as two component models. The first is a *Bayesian decision model*, which operates at the lower-level of the decision process. This probabilistic model acts as a filter between the monitoring system and the reallocation module to reduce false detection and incorrect decisions. This will increase the stability of the application and reduce the amount of unnecessary overhead incurred by reallocation in response to false detections. The second is a *Markovian decision model* which operates at a higher level of the decision process. The Markovian model is designed to keep track of global application performance by monitoring the state transitions of various performance metrics. Using the state transition data, the Markovian model is able to predict the steady-state system performance and react to potential future performance degradations.

3 Decision Models

The simplest decision model in our framework (Figure 1) has been optimized for low latency rather than high accuracy decisions. In this model, referred to as the *baseline model*, any detection event immediately triggers a cost evaluation and potential reallocation. It is apparent that this decision model can lead to unnecessary overhead, in part because the cost evaluation penalty is incurred *every* time a detection event occurs. Often, because of imperfect monitors, a detection event is not indicative of the overall performance falling beneath the required limit. In this situation, referred to as *false detection*, the overhead involved in cost evaluation is incurred with no potential for benefit. In response, we propose a two-level decision model for such systems. The first model is an adaptation of a Bayesian decision model originally formulated by Nicol [2]. The second is a Markovian decision model formulated to predict global trends in application performance.

3.1 Bayesian model

There are several sources of difficulty that can make the baseline decision model ineffective. First, the application may be subject to transient load spikes. Since the application is data dependent, it is sensitive to any changes in the input stream. A problem occurs with noisy data or "spikes" that represent transient load changes rather than a stable shift in computational loads. In such situations, a detection event may not be indicative of a longer term change in load, necessitating a resource reallocation. Rather, it may represent a transient condition, in which case a reallocation may actually negatively affect performance. In terms of our ATR example, changes in scenery can result in input spikes. In the absence of new targets, these spikes are transient and should not be grounds for reallocation. An effective resource management system must be capable of discerning stable computational shifts from spikes. Second, the detectors themselves possess some degree of unreliability. It is possible for detectors both to generate a false detection event or fail to detect a genuine detection event. Increasingly accurate detectors are computationally intensive and increase the latency between the occurrence of a load imbalance and corresponding detection. On the other hand, simple load detectors may not be effective in accurately detecting stable load changes. They can significantly raise computation overhead by increasing the number of false detections. The proposed Bayesian decision model will allow for quick detection of critical events using simpler detectors while reducing reports of false detections.

3.1.1 Model description

Faced with computational overhead and potentially poor performance in the event of an unnecessary reallocation, it is not beneficial to run the cost evaluator on the basis of a single positive report. As illustrated in Figure 2, the Bayesian decision model adds an extra component, operating as a *smart filter*, to the system. In this proposed configuration, a collection of monitors still record execution parameters as frames pass through the application. However, in this scheme, all detection events pass through the smart filter. The smart filter uses a Bayesian decision model to determine if the cost evaluator should be invoked. It is the goal of the smart filter to minimize the number of false detections passed to the cost evaluator. Ideally, the cost evaluator will only be signaled if a potential remapping benefit is very likely. Conversely, whenever a remapping benefit becomes likely, the cost evaluator should be signaled as soon as possible.

At each frame time, we compute the probability that performance can be improved by reallocating resources. This probability is referred to as the *gain probability* and is represented by the symbol p_n [2]. The Bayesian decision process must constantly strengthen or weaken the gain probability based only on information from the detectors and information about the quality of their detections. These detectors operate on the instrumented streams returned from the monitors and look for various metrics at the task level. While the overall application behavior may be within real-time bounds, the detectors can notice if a specific task's performance starts to decrease. By operating at the lower level, this model can determine improved resource mappings even if the real-time requirements are not immediately threatened.

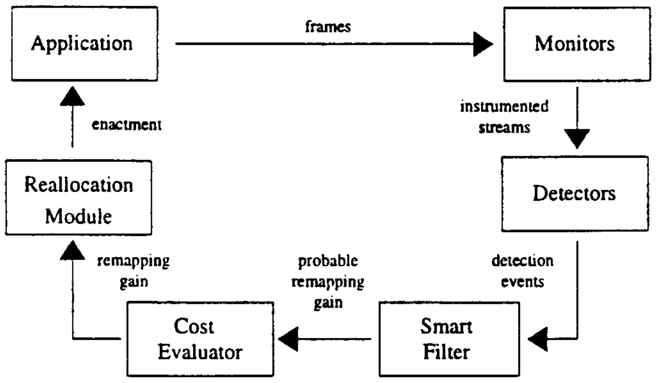


Figure 2: Block diagram of the Bayesian decision model.

3.1.2 Model calculation

To implement this Bayesian model, three distinct parameters must be defined [2].

1. ϕ – The probability that a performance gain can be realized by remapping after the current frame, given that it has not been realizable in the previous frame.
2. α – The probability of the detectors prematurely reporting a remapping condition.
3. β – The probability of the detectors failing to report an existing remapping condition.

The parameters α and β encompass the fact that the detectors contain some inherent inaccuracy. Based on the above parameters, we calculate the probability p_n that a performance gain can be realized on frame n based upon the results returned by the detectors.

First, we define the *pretest probability*, p_a, that a performance gain is achievable on frame n, given that $p_{n-1} = p$:

$$p_a(p) = p + (1-p)\phi$$

Bayes' Theorem states that for two events A and B, the probability of A given B is:

$$P(A \mid B) = \frac{P(B \mid A) \cdot P(A)}{P(B \mid A) \cdot P(A) + P(B \mid A^C) \cdot P(A^C)}$$

Now, if the detectors return a positive remapping report, we want to calculate the probability that an actual performance gain exists. Using the above notation, A is the event where a performance gain exists and B is the event of a positive report. Therefore, $P(A)$ is given by the pretest probability, p_a, and $P(B \mid A)$ is given by the probability of an accurate positive report. Since β is the probability of the detectors failing to report a gain condition, $(1-\beta)$ is the probability that the detectors accurately report a gain condition. $P(A^C)$ is the complement of $P(A)$, which is given by $(1-p_a)$. Lastly, $P(B \mid A^C)$ is the probability that a positive report is returned given that no remapping gain exists. Recall that this is the exact definition of the parameter α. Substituting these expressions into Bayes' Theorem gives us the following gain probability for a positive report:

$$p_n = \frac{(1-\beta) \cdot p_a(p)}{(1-\beta) \cdot p_a(p) + \alpha \cdot (1 - p_a(p))}$$

Using similar logic, the gain probability in the event of a negative report can be computed as follows:

$$p_n = \frac{\beta \cdot p_a(p)}{\beta \cdot p_a(p) + (1-\alpha) \cdot (1 - p_a(p))}$$

In either case, this probability, p_n, represents a weighted measure of the potential for a remapping gain to exist after frame n. When p_n crosses a suitable threshold level, it is deemed likely that a remapping gain exists. This in turn justifies a cost evaluation to determine if a better resource allocation is realizable. If so, the reallocation module is signaled and the remapping is enacted.

3.2 Markovian model

Bayesian decision models are sensitive to the accuracy of model parameters and as such are implemented at a low level within the application. These models tend to be *reactive* as they wait for events signaled by the detectors. At a higher level of abstraction, we can track the application performance based on the degree to which user specified performance bounds are met. Toward this end, we formulate a Markovian decision model. This model is *predictive* as it uses performance state data to predict the future behavior of the application. In terms of our ATR application, the Markovian model analyzes scenic trends. While an individual scene may not provide much information, a collection of scenes can provide insight about future direction and the resources that may be required by specific tasks. The Markovian model uses these trends to predict the need for a resource reallocation before it actually occurs.

3.2.1 Model description

By working at the lower level, the Bayesian model is able to detect improved resource mappings even when the application is conforming to the real-time specifications. The Markovian model, which operates at a higher level, is triggered solely by the level of conformity with the real-time bounds. It will only trigger a remapping if the application is predicted to violate these bounds with a high probability. The Markovian decision model can be viewed as a *watchdog* for the Bayesian model. If the Bayesian model is able to maintain performance within the desired specifications, the Markovian model will never intercede. However, if the system performance appears to threaten the real-time specifications, the Markovian model will override the Bayesian model and force a resource reallocation. An example of this process is presented in Section 3.2.2.

Figure 3 presents a block diagram of the system incorporating the Markovian decision model. Its primary function is to monitor specific evaluation metrics of global application performance, e.g. end-to-end frame latency. This is done by comparing actual measured statistics with the real-time specifications. The ratio of the measured statistic to the desired bound serves as a metric of the level of conformity of the application. This level of conformity is then used to map the application into one of a set of previously defined *performance states*. These performance states and the transitions between them provide the underlying framework of the Markovian model.

The inherent differences between the two decision models enable them to be coupled in a synergistic manner. The Bayesian model is constantly checking the detection streams and updating the gain probability. When the gain probability exceeds a threshold, it signifies a high potential for a remapping gain. To take full advantage of this potential and reduce the chance of missed deadlines, the Bayesian model is coupled with a simple reallocation

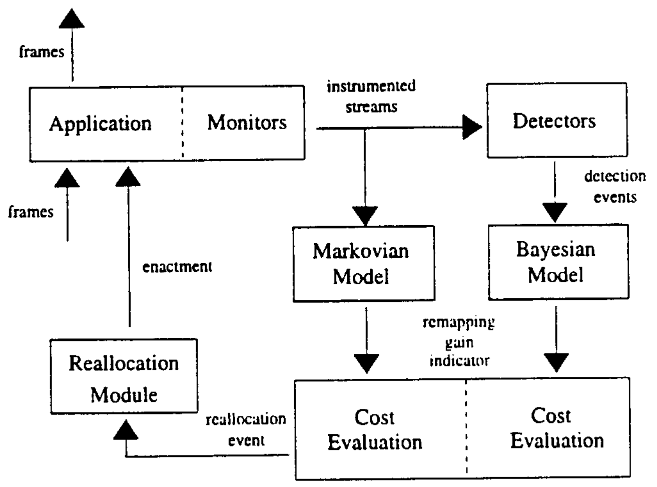


Figure 3: Block diagram of the coupled decision model.

algorithm to provide a low latency solution. While these decisions allow a quick response and often provide an immediate improvement, they may not represent the best resource allocation. To correct for this, the Markovian model is invoked at periodic intervals. If the predicted performance of the application is sufficiently poor, a (relatively) more complex reallocation algorithm can be invoked. This algorithm performs a more extensive (and therefore costlier) assessment of remapping potential. In these instances, a high quality remapping which can provide system-wide improvement is more effective than a high speed decision which can provide immediate but local improvement.

3.2.2 Model calculation

We first provide an intuition about the application of the model through the following example. Consider the state space of an ATR application as represented in Figure 4 where we wish to maintain a frame analysis rate above 33 frames/sec. Therefore, we must maintain end-to-end frame latencies below 0.03 seconds. The maximum frame latency we expect is .10 seconds and the latency range is divided into five states. States 0, 1, and 2 are considered acceptable and states 3 and 4 are considered unacceptable.

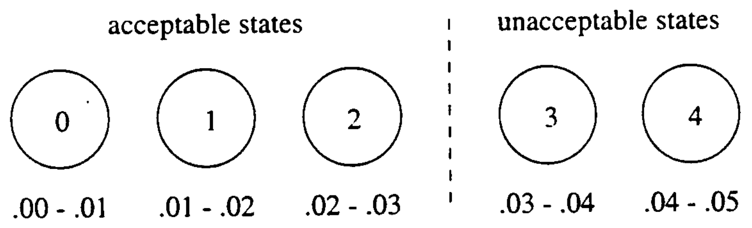


Figure 4: A high level example of the Markovian decision process.

Over time, frames are periodically generated by the source and injected into the system. At specific instances, referred to as *frame intervals*, completed frames are consumed at the sink. The time between frame generation and consumption is the end-to-end frame latency which we are trying to control. At each frame interval, the monitored information streams can be used to generate a current snapshot of the application in terms of frame latency performance. This snapshot is used to categorize the system as residing in a particular "performance state." As shown in Figure 4, the range of performance states is easily partitioned into a collection of acceptable and unacceptable states. As the data content of the input frames varies over time, the current performance state of the system will fluctuate. Resource reallocation is used to improve the performance of the system when it resides in an unacceptable state. The goal of the Markovian model is to predict when the system is moving toward an unacceptable state and to avoid it by triggering a resource reallocation in advance. This is accomplished by recording statistics about the state transitions experienced by the application. Using the state transition information and Markov theory, we can predict the steady-state behavior of the application. At periodic intervals, this steady-state prediction is used to determine if resource reallocation is necessary. This interval is referred to as the *Markov invocation interval.*

The following description is based on sensor applications where data frames are arriving at some rate. The model can be generalized in a straightforward manner to more general event driven applications [4] where the events such as a frames may arrive asynchronously rather than at a fixed rate.

Given the number of states (n), the maximum performance measurement (μ) and the current performance measurement (ρ), the current state is determined by:

$$\frac{\rho \cdot n}{\mu}$$

To ensure proper operation, any performance measurement exceeding μ is automatically categorized into the largest state. Note that in this model, performance metrics are mapped to states numbered from 0 to $n-1$. There is a natural mapping for metrics such as latency, since higher latency values map to higher numbered states. The mapping may be different for metrics such as throughput where lower throughput values map to higher numbered states so that we have a consistent interpretation of higher numbered states corresponding to lower performance.

The Markovian decision model tracks the efficiency of the current mapping by calculating the performance state of the system as described above. These performance states can be viewed as a Markov chain, with the application transitioning between them at each frame time based upon the current data and resource mapping. As the application moves between the performance states, the Markov model maintains statistics about the state transitions. With this information, at any point during execution, the Markov model has an accurate picture of the state transition probabilities of the Markov chain. These transition probabilities are conditional probabilities for the system to transition to a particular state, given the current state of the system. If there are n states in the Markov chain. than the collection of all possible one-step transitions can be collected in an $n \times n$ matrix called the *state transition matrix.* This state transition matrix can then be used to calculate the the *steady-state probability vector:*

$$\mathbf{S} = [p_0 \; p_1 \; p_2 \; \cdots \; p_{n-2} \; p_{n-1}]$$

The steady-state probabilities predict, based on the current picture of the system, the probability of the system settling in each state. This provides a measure of the long-term global application behavior. While the Bayesian attempts to make locally optimal decisions, the steady-state probabilities may show that the system is heading towards an increasingly unbalanced state. The steady-state probabilities are examined to determine whether reallocation is necessary. We use the following approach which accounts for the gradient of unacceptable states. A detailed description of the model can be found in [7].

Using the steady-state vector, the following inner product is computed.

$$\omega = \sum_{i=1}^{n} p_i \cdot i$$

The above computation produces a weighted state, ω. If this weighted state falls within the set of undesirable states, then a reallocation is invoked. By basing its decisions upon the steady-state probabilities, the Markovian decision model represents a *predictive* process, while the Bayesian decision model represents a *reactive* process.

3.3 Model Parameters

For these decision models to correctly determine when a remapping gain is probable, they must have accurate knowledge of certain application-specific parameters. For the Bayesian model, these are ϕ, α, and β as discussed earlier, and they are measures of how often a remapping gain is achievable and how accurately the detectors can recognize this situation. For the Markovian model, the number and range of performance states and the threshold between acceptable and unacceptable states must be defined. In addition, the statistics for transitions between these states and the duration of the interval between Markov invocations are necessary for calculating the steady-state distribution.

A detailed explanation of the experiments to determine these parameter values can be found in [7]. For our experiments, the following values were used for the Bayesian parameters:

$$\alpha = 0.125037 \quad \beta = 0.34588 \quad \phi = 0.09500$$

Our experiments use Markov chains consisting of 20 states representing the frame latency. To provide a reasonable tolerance for an occasional missed deadline and a stricter tolerance for any consecutive misses, the Markov demarcation threshold was chosen to be 12. The frequency of change in the input stream is characterized by a parameter called the *stability interval.* The stability interval refers to the number of frames over which the application remains stable, i.e., does not require reallocation. As the Markovian decision model is essentially *sampling* the input stream at each invocation, this translates to a Markovian invocation interval of half the length of the input stability interval.

4 Performance Evaluation

This section compares the performance of the decision models under varying input conditions. Our experimental platform consisted of two components. An 8-node IBM SP-2 was used for our empirical studies which generated the application specific model parameters. The "ATR application" was a synthetic workload generator that can be configured to represent a range of ATR workloads. Simulations were run with the synthetic workload generator running on a uniprocessor. Our synthetic workload generator allows us to compare the different models while changing a number of important input characteristics. The two primary input characteristics we investigated were rate of change of workload and the presence of noise. Rate of change refers to how frequently the input frames cause substantial workload changes in the tasks of the application requiring reallocation. Noise refers to both the frequency and size of input workload spikes. These parameters were chosen because they are most applicable to the types of event-driven applications that these models are designed to improve. They are also characteristics for which values can often be ascertained a priori. For example we may know the maximum rate at which new targets can appear in the scene or we may know something about he quality of the detectors and sensors, or be familiar with texture of the terrain. The factors affect the presence of workload spikes and rate at which we can expect stable shifts in workload.

Three sets of experiments were performed in this section. The first set of experiments were designed to evaluate the improvement of the Bayesian decision model over the simple model used as our baseline. The second set of experiments were designed to illustrate the predictive capabilities of the Markovian decision model and its ability to improve application performance when used in conjunction with the Bayesian model. The final set of experiments test the fully coupled Bayesian and Markovian models under various input conditions characterized by their rate of change and the presence of noise. For comparison purposes, the baseline decision model refers to the model where every detection event invokes a cost evaluation.

4.1 Bayesian model

The following graphs illustrate performance improvements provided by the Bayesian model over the baseline decision model. Figure 5 demonstrates the reduction in false detection percentage achieved by the Bayesian model. Figures 6 and 7 indicate the end-to-end latency of the injected frames and the number of resource reallocations required to achieve that latency using each decision model. These graphs provide a comparison of the number of resource reallocations enacted by the decision models in response to identical input streams. Figure 8 plots the end-to-end frame latencies for both models on the same axes. This graph is used to compare the overall performance of the two models.

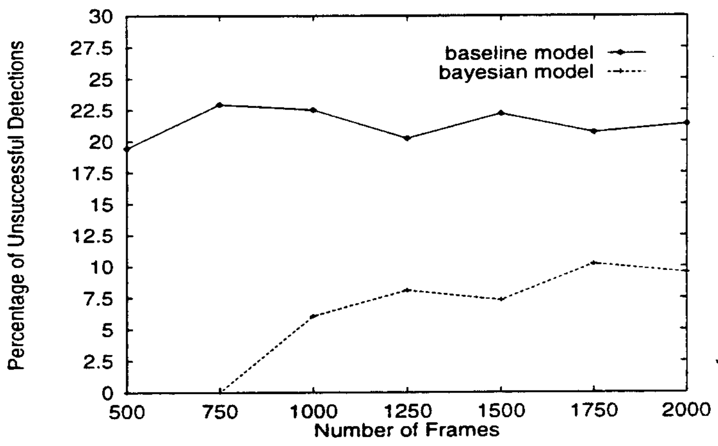


Figure 5: An illustration of the false detection rate for the two decision models.

It is apparent from Figure 5 that the Bayesian decision model significantly reduces the percentage of unnecessary invocations of the cost evaluator. The smart filter is able to successfully pare the number of detection events that will not result in a remapping gain, thereby reducing the amount of unnecessary cost evaluation overhead. Furthermore, by decreasing the the total number resource reallocations, the Bayesian model also reduces the amount of unnecessary reallocation overhead. This is accomplished by filtering the detectors response to noise and input spikes. The baseline decision model often reacts to input spikes by reallocating resources at frames where the spike is detected and on immediately successive frames. This behavior represents unnecessary reallocation overhead as the input spikes are transient and provide very limited rewards following a resource reallocation.

The combination of reducing both false detection percentage and unnecessary reallocations improves the behavior of the application in a number of ways. First, less computation time is spent in the cost evaluation module, and ultimately more processing power can be used for useful computation. Second, by ensuring that the cost evaluator is invoked only when there is a high probability for a remapping gain, a more complex evaluation mechanism can be enacted with (relatively) less overhead. The following two figures compare the number of resource reallocations required by the baseline and Bayesian models and the latency characteristics under which the reallocations occur.

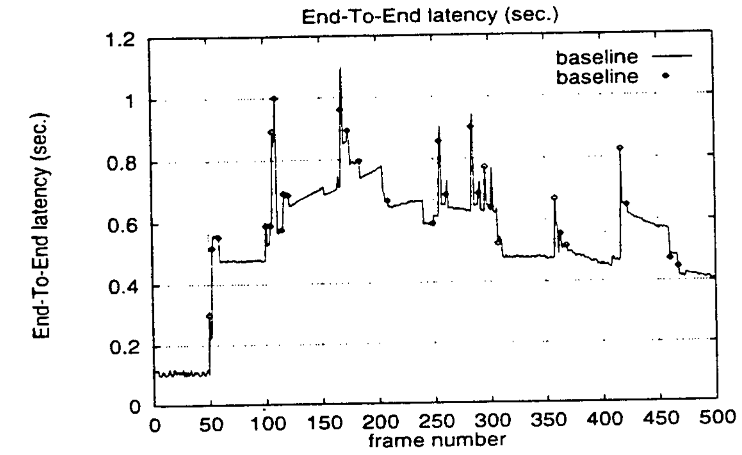


Figure 6: An illustration of frame latency and reallocation points for the ***baseline model.***

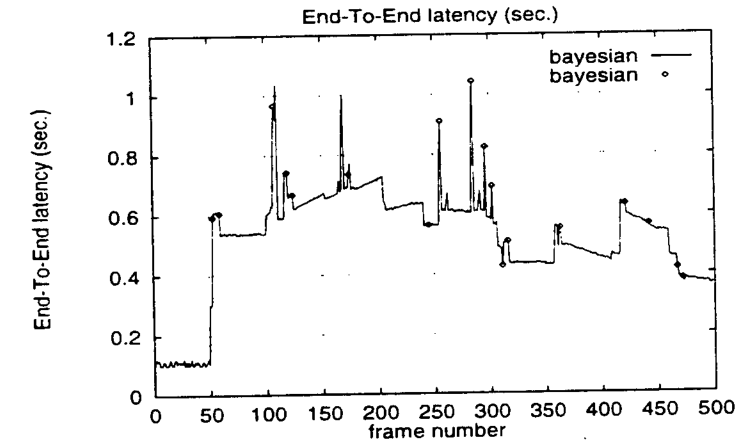


Figure 7: An illustration of frame latency and reallocation points for the ***bayesian model.***

To fully understand the benefits of the Bayesian decision model, one must look at the remapping behavior of the system along with the frame latency through the system. Figure 6 shows that the baseline decision model remaps a total of 29 times, while Figure 7 shows that the Bayesian decision model remaps a total of 18 times. This reduction in the number of resource reallocations primarily results from the filtering of detection events directly associated with the presence of input spikes.

Because of their transient nature, input spikes cause detection events based upon conditions that do not persist after the short duration of the spike. Spikes can lead to new resource allocations predicated on information which is not indicative of a long-term behavioral change. Resource reallocations resulting from input spikes do not provide large enough performance benefits to justify their enactment overhead. However, given that an input spike has already caused a remapping, it may not be necessary to automatically remap after the spike to correct the situation. A spike-based reallocation may not provide an increase in performance, but it also may not perturb the system enough to cause a decrease in performance. It is not always beneficial to reallocate resources immediately following an input spike because the benefits may not outweigh the enactment overhead. Because input spikes produce easily detectable changes both on the way up and on the way down, the simple decision model often invokes a resource reallocation both before and after the spike. The smart filter embedded in the Bayesian decision model allows it to reduce the number of initial reactions to input spikes. Despite this, large or long-lasting spikes will cause the Bayesian model to react. In these situations, the Bayesian model may also filter post-spike reactions if the performance benefits are not significant enough to warrant a resource reallocation. By utilizing both pre- and post-spike filtering, the Bayesian model reduces the total number of resource reallocations and the effect of their enactment overhead on frame latency. Figure 8 shows a comparison of end-to-end frames latency for the baseline and the Bayesian decision models.

Figure 8 demonstrates that the Bayesian model provides consistently improved latency performance over the course of the application's execution. The Bayesian decision model spends less time reacting to input spikes and makes smarter decisions than the baseline model. The benefits of the Bayesian decision model are twofold. First, it can provide improved end-to-end latency performance during execution. Second, it significantly reduces the number of resource reallocations necessary to provide this performance.

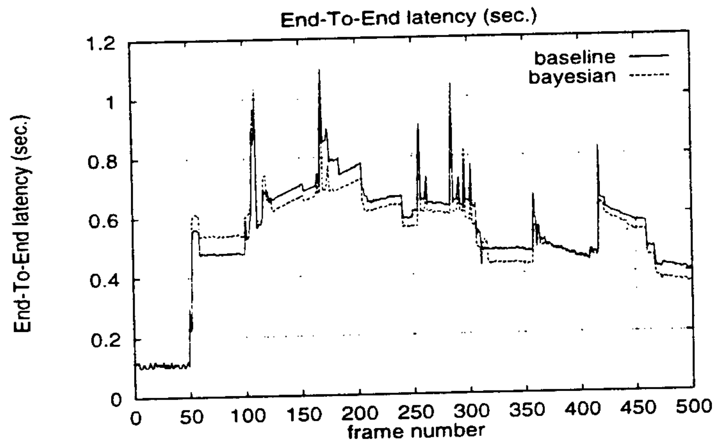


Figure 8: A comparison of frame latencies for the baseline and Bayesian models

The ability to filter the effects of input spikes is at the heart of the Bayesian model's improved performance. A trade-off exists between filtering input spikes and reacting to real load changes. As shown in Figure 7, the Bayesian model is not able to filter very large or prolonged input spikes. Attempting to filter these conditions could result in a large number of real input load changes being undetected. This potential is illustrated in Figure 8 during frames 50 through 100. This is the only case in our tests for which the simple model provided better performance than the Bayesian model. In this situation, the Bayesian model filtered a detection event signifying a stable load change and was therefore slow to react to the actual conditions. The profiling scheme used to fine tune the Bayesian parameters to this application limits the occurrence of this situation to an initial startup transient of the Bayesian model or after extended periods of stable input with no activity. An important feature of the Markovian model is its ability to detect these remaining occurrences and reduce their effect to negligible levels.

4.2 Addition of the Markovian model

This section presents the results of coupling the Bayesian and Markovian decision models. Figure 9 demonstrates the potential of the coupled model over the pure Bayesian model.

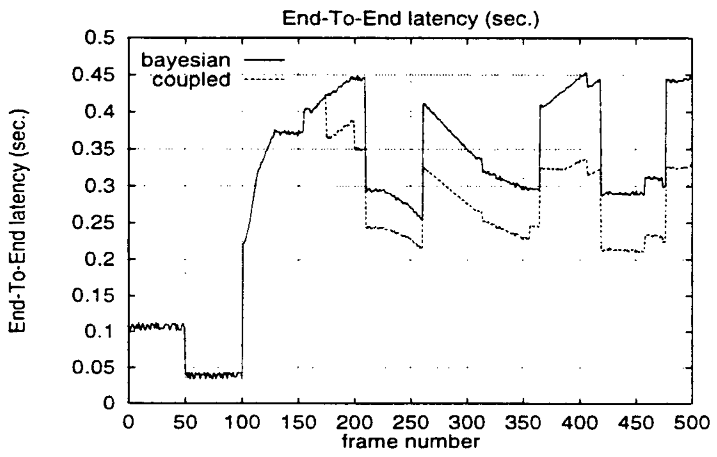


Figure 9: A comparison of frame latencies between the *Bayesian model* and the *coupled model*

As illustrated, the two experiments show similar performance until frame 175. Up to this point, the application has been conforming to the real-time specifications and the Markovian model invocations have continually predicted acceptable performance. All resource reallocation have resulted from the Bayesian models. At frame 175, the coupled system invokes the Markovian model and it predicts that system performance is heading toward an unacceptable state. This results in a remapping which is represented by the latency drop in Figure 9. Following this decision, the coupled model has a better resource allocation than the pure Bayesian model. This is confirmed by the lower end-to-end frame latency shown in Figure 9. For the next 175 frames, the Markovian model is again inactive as the application is within the real-time bounds. Both systems are again making purely Bayesian decisions, which accounts for the similarity in the shapes of the two graphs. When the Markovian model is invoked at frame 350, it again predicts that the system is heading toward an unacceptable state. Another Markovian reallocation occurs, which accounts for the slight deviation in the shape of the graphs between frames 350 and 400. The pure Bayesian system shows an increasing latency slope beginning around frame 360, while the coupled system does not show an increase in latency until around frame 385.

These results clearly show the ability of the Markovian model to monitor the global application performance and initiate a new resource allocation when the real-time specifications are threatened. This predictive capacity makes the Markovian model perfectly suited to act as a watchdog over the Bayesian model.

4.3 Coupled model performance

This section presents a final set of results. We have demonstrated both the benefits of the Bayesian model over the baseline model and the benefits of adding a Markovian watchdog to the Bayesian process. We now investigate the performance of the coupled decision model with respect to two important input parameters: input rate of change and noise. These two conditions represent important dimensions of the applications that may utilize adaptive resource management. For our ATR application, we want to maintain an acceptable frame rate in the presence of both a high number of targets (input rate) and a large amount of scenic variations (input noise).

A particular input stream can be described in terms of the range of rate of change and noise. We studied the performance of the coupled model under average and extreme conditions as illustrated in in Figure 10.

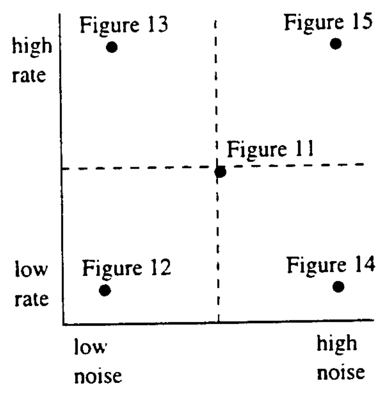


Figure 10: A representation of the input characteristics for our experiments and reference to the corresponding figures.

Our first experiments consisted of testing the coupled decision model on "average" input streams with median values for rate of change and noise. The input streams used in these experiments have stability intervals on the order of 50 frames and input spike probabilities on the order of 15 percent. Performance results from these experiments comparing the behavior of the coupled and baseline models are provided in Figure 11.

As evidenced in Figure 11, the coupled decision model is able to improve end-to-end frame latency throughout the course of execution. Under conditions containing median levels for both input rate of change and noise, both the Bayesian and the Markovian models contribute toward improved performance. By reducing the total number of decisions and improving decision quality and timing, latency performance is improved while decision and enactment overhead is reduced.

The following two experiments investigate input streams with a low probability of input noise. The average probability of input spikes used in these experiments was 5 percent. We further divided these experiments into input streams containing either a high or low rate of input change. Low rate of change input streams used stability intervals on the order of 100 frames, while high rate of change input streams used stability intervals on the order of 20 frames. Figure 12 compares the results from the low noise and low rate of change experiments, and Figure 13 compares the results from the low noise and high rate of change experiments.

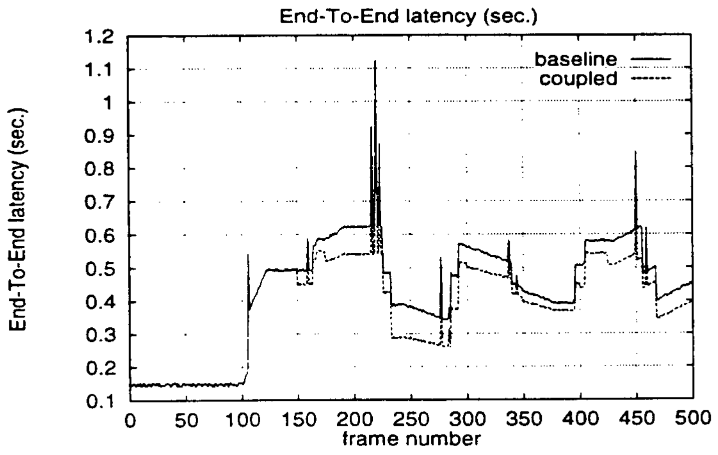


Figure 11: A comparison of frame latencies between the baseline model and the coupled model under *average* input conditions.

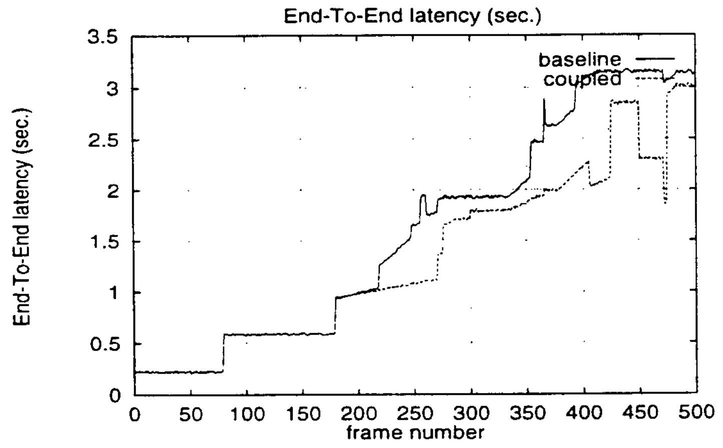


Figure 12: A comparison of frame latencies between the baseline model and the coupled model with *low rate* and *low noise*.

Under low noise conditions, the coupled decision model performs significantly better than the baseline decision model. The reduction in input spikes resulting from the low noise behavior reduces the dependence on the Bayesian smart filter. In these experiments, the Markovian model contributes more significant decisions than the Bayesian model. Low levels of noise increase the accuracy of the Markovian predictions. This allows the Markov model to make better decisions about when and where to best allocate resources. Figure 12 demonstrates that good Markovian decisions can significantly improve the performance in a low rate of change input stream. Because of the low rate of change, both the baseline and Bayesian models do not receive many detection events and therefore do not trigger/enact many reallocations. The Markovian model is able to push the system into a more globally efficient state which persists because of the low input variation. Figure 13 demonstrates that in a low noise environment the coupled decision model is also effective for a high rate of input change. By adjusting the sampling frequency of the Markovian model to account for the increased input data rate, the coupled model is able improve the performance of the application over a large number of frames.

The final two experiments investigate input streams with a high probability of input noise. The average probability of input spikes used in these experiments was 50 percent. We further divided these experiments into input streams containing either a high or low rate

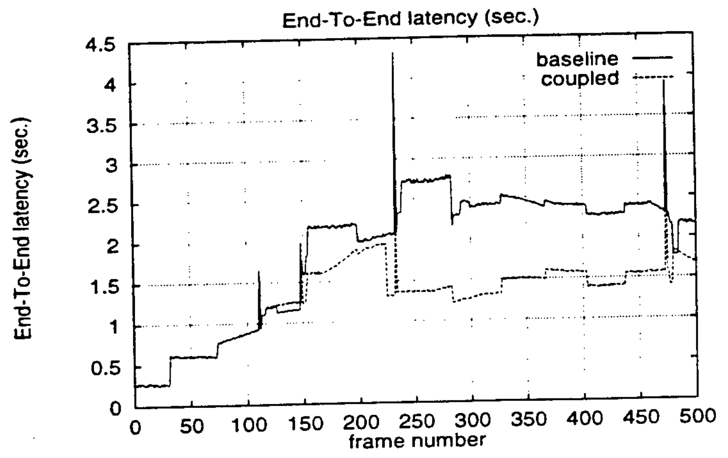


Figure 13: A comparison of frame latencies between the baseline model and the coupled model with *high rate* and *low noise*.

of input change. Low rate of change input streams used stability intervals on the order of 100 frames, while high rate of change input streams used stability intervals on the order of 20 frames. Figure 14 compares the results from the high noise and low rate of change experiments, and Figure 15 compares the results from the high noise and high rate of change experiments.

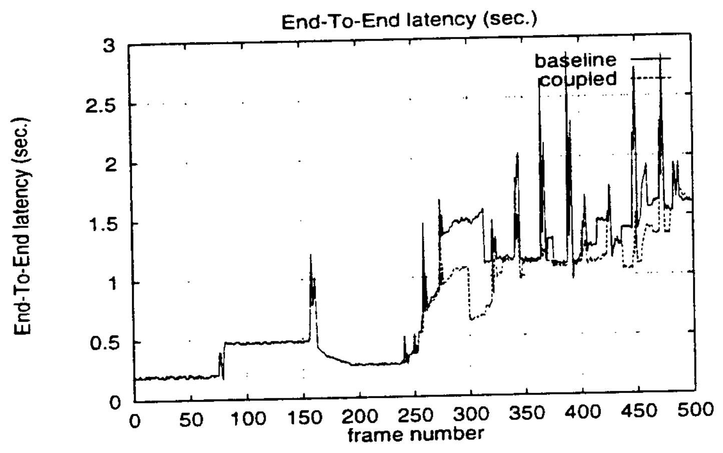


Figure 14: A comparison of frame latencies between the baseline model and the coupled model with *low rate* and *high noise*.

Because of the high levels of noise in these two experiments, the improvements in end-to-end frame latency are not as prominent as in previous experiments. Under these input conditions, the Bayesian model is more effective than the Markovian model. The frequency of the input spikes limits the accuracy of the Markovian predictions. The increased noise also results in an greater number of lower-level resource reallocations. These additional reallocations ensure that the Markov statistics are initialized more frequently, thereby limiting the model's effectiveness as a predictor. However, the Markovian model does serve an important purpose in these experiments. As the embedded smart filter in the Bayesian model attempts to filter out the input noise, the Markovian model acts as a backup to ensure that it does not filter any significant load changes. Any sustained performance levels violating the real-time specifications will still be corrected by the Markovian model.

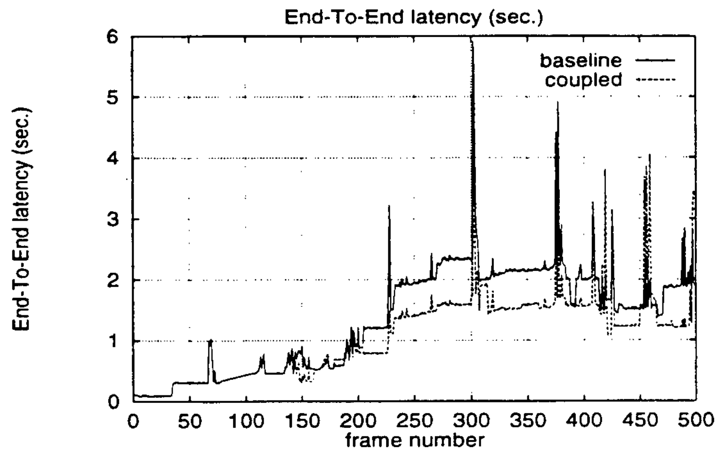


Figure 15: A comparison of frame latencies between the baseline model and the coupled model with *high rate* and *high noise*.

While these figures demonstrate improved latency performance for the coupled decision model, additional benefits are not apparent from the graphs. The filtering properties of the Bayesian model along with the Markovian backup allow the coupled model to achieve better latency performance in a fraction of the cost evaluations and resource reallocations required by the baseline model.

Our experiments demonstrate the effectiveness of the coupled model across a range of input conditions. The total number of resource reallocations and the number of false detections are both significantly reduced. These reductions are accomplished while maintaining improved latency performance. The reduction in both detection and enactment overhead allow more processing cycles for useful work without sacrificing any latency performance. This is significant in that these experiments are not based on a fixed number of processor cycles distributed between useful computation and allocation. In a practical implementation with real applications, we expect that the latency improvements using a fixed number of processors will be greater since the cycles saved by effective detection and prediction will directly reduce the latency.

5 Conclusions and Future Research

Clearly, there is a need for efficient adaptive resource mechanisms to be used with data-dependent, real-time applications. The mechanisms must be responsive to change and yet accurate in their remapping requests. These quality requirements place a great deal of pressure on the remapping decision model. Current implementations of simple decision models might not be able to meet increasingly stringent real-time requirements. This paper proposes an improved decision process to provide increasingly accurate resource mappings while maintaining low decision latency and overhead.

Experiments using a synthetic workload generator and the statically defined model parameters yielded promising results in multiple categories. An overall reduction in the percentage of unsuccessful invocations of the cost evaluator and number of unnecessary resource reallocations was realized with the Bayesian decision model. This allows more cycles for useful computation and can mask the use of the more complex Markovian decision process. Experiments with frame latency showed similar or improved performance compared with the simple decision model for a significantly lower number of remappings.

By coupling the reactive Bayesian model with the predictive Markovian model, we create a multi-level decision model capable of improving the performance of adaptive resource managers under a variety of input conditions. Under average input conditions, both models contribute to decrease the end-to-end latency of input frames and reduce the decision and enactment overhead. Toward the extremes, the Bayesian model proves more applicable to high noise environments and the Markovian model better suited to low noise environments. In these situations, the less suited model provides good backup support for the more effective model. Under low noise conditions, the Bayesian level keeps track with the baseline model while the Markovian level pushed the system toward more acceptable performance states. Under high noise conditions, the Bayesian level filters a much larger percentage of the input spikes while the Markovian level ensured performance did not fall below the real-time specifications. Over a wide range of input streams, the coupled model is shown to maintain or improve the latency performance while decreasing the number of false detections and unnecessary resource reallocations.

Future work in the context of this system will include methods for dynamically varying the Bayesian and Markovian thresholds in response to the current task-level resource allocation. We also plan to implement mechanisms allowing the Markovian model to suggest appropriate resource allocations for the steady-state behaviors it currently predicts. In addition, we are currently working on an 3-D tracking system that will allow us to test these decision models in the framework of an actual application.

6 Acknowledgment

The authors would like to thank Daniela Ivan Roşu for her work developing the Adaptive Resource Management framework we used to test our models, and her invaluable help in the use and application of this framework.

References

[1] R. Jha, M. Muhammad, S. Yalamanchili, D. Ivan-Rosu, C. de Castro, "Adaptive Resource Allocation for Embedded Parallel Applications," *Proceedings of the Third International Conference on High Performance Computing Systems*, 1996.

[2] D. Nicol and P. Reynolds Jr., "Optimal Dynamic Remapping of Data Parallel Computations", *IEEE Transactions on Computers*, February 1990.

[3] D. Nicol and J. Saltz, "Dynamic Remapping of Parallel Computations with Varying Resource Demands", *IEEE Transactions on Computers*, September 1988.

[4] D. Ivan-Rosu, K. Schwan, S. Yalamanchili, R. Jha, "On Adaptive Resource Allocation for Complex Real-Time Application," *Proceedings of the Real Time Systems Symposium*, December 1997.

[5] B. Mukherjee and K. Schwan, "Improving Performance by Use of Adaptive Objects: Experimentation with a Configurable Multiprocessor Thread Package", *Proceedings of Second International Symposium on High Performance Distributed Computing*, July 1993.

[6] B. Schroeder, G. Eisenhauer, K. Schwan, J. Heiner, V. Martin, and J. Vetter, "From Interactive Applications to Distribute Laboratories", to appear in *IEEE Concurrency*, 1997.

[7] D. Paul, "Decision Models for On-line Adaptive Resource Management", a Masters Thesis to be presented to the faculty of the Georgia Institute of Technology in November, 1997.

[8] Y. C. Chang and K. G. Shin, "Optimal Load Sharing in Distributed Real-Time Systems," *Journal of Parallel and Distributed Computing*, pp. 38-50, 1993.

[9] D. L. Eager, E. D. Lazowska, and J. Zahorajan, "Adaptive Load Sharing in Homogeneous Distributed Systems," *IEEE Transactions on Software Engineering*, May 1986.

[10] S.H. Bokhari. Partitioning problems in parallel, pipelined and distributed computing. *IEEE Transactions on Computers*, Jan. 1988.

[11] Honeywell, Inc. C3I parallel benchmark suite. Technical Information Report, Aug. 1996.

[12] J. Huang and W.P.-J., "On supporting mission-critical multimedia applications," *Proceedings of the 3rd IEEE International Conference on Multimedia Computing and Systems*, 1996.

[13] C. McCann and J. Zahorjan, "Processor allocation policies for message-passing parallel computers," *Proceedings of ACM Sigmetrics*, 1994.

[14] K-H. Park and L.W. Dowdy, "Dynamic partitioning of multiprocessor systems," *International Journal of Parallel Programming*, No. 2, 1989.

[15] K. Ramamritham and J.A. Stankovic, "Dynamic task scheduling in hard real-time distributed systems'," *IEEE Software*, Vol. 1, No. 3, pp. 65-75, July1984.

[16] D.I. Rosu and K. Schwan, "Improving protocol performance by dynamic control of communication resources," *Proceedings of the 2nd IEEE ICECCS*, 1996.

[17] H. Rotithor and S. Pyo, "Decentralized decision making in adaptive task sharing," *Proceedings of the 2nd IEEE Symposium on Parallel and Distributed Processing*, 1990.

[18] T.Bihari and K. Schwan, "A comparison of four adaptation algorithms for increasing the reliability of real-time software," *Proceedings of the Real-Time Systems Symposium*, 1988.

[19] T.Bihari and K. Schwan, "Dynamic adaptation of real-time software," *ACM Transactions on Computer Systems*, May 1991.

[20] A. Tucker and A. Gupta, "Process control and scheduling issues for multiprogrammed shared-memory multiprocessors," *12th ACM Symposium on Operating Systems Principles*, 1989.

[21] H. Zhou, K. Schwan, and I. Akyildiz, "Performance effects of information sharing in a distributed multiprocessor real-time scheduler," *Proceedings of the Real-Time Systems Symposium*, 1992.

Hierarchical Architecture For Real-Time Adaptive Resource Management

Authors:	Ionut Cardei, Rakesh Jha, Mihaela Cardei, Allalaghatta Pavan
Primary Contact:	Ionut Cardei
Email:	icardei@htc.honeywell.com
Address:	Honeywell Technology Center, MN65-2600 3660 Technology Drive Minneapolis, MN 55418, USA Tel: (612)-9517138 Fax: (612)-9517438

ABSTRACT

Mission-critical distributed applications must be able to adapt to mission dependent variations in resource demands as well as dynamic changes in resource availability. A middleware layer designed to provide QoS-aware resource management facilitates application development and follows the current industry trend towards cost-effective COTS-based implementations. This paper presents the Real Time Adaptive Resource Management system (RTARM[1]), developed at the Honeywell Technology Center. The RTARM system supports provision of integrated services for real-time distributed applications and offers services for end-to-end QoS negotiation, QoS adaptation, real-time application QoS monitoring and hierarchical QoS feedback adaptation. In this paper, we focus on the hierarchical architecture of RTARM, its flexibility, internal mechanisms and protocols that enable management of resources for integrated services. The architecture extensibility is emphasized with the description of several service managers, including an object wrapper build around the NetEx real-time network resource management system developed by the Texas A&M University. We use practical experiments with a distributed Automatic Target Recognition application and a synthetic pipeline application to illustrate the impact of RTARM on the application behavior and to evaluate the system's performance.

Key words: adaptive resource management, distributed real-time applications, integrated services, QoS negotiation and adaptation, hierarchical feedback adaptation

Hierarchical Architecture For Real-Time Adaptive Resource Management

ABSTRACT

Mission-critical distributed applications must be able to adapt to mission dependent variations in resource demands as well as dynamic changes in resource availability. A middleware layer designed to provide QoS-aware resource management facilitates application development and follows the current industry trend towards cost-effective COTS-based implementations. This paper presents the Real Time Adaptive Resource Management system (RTARM), developed at the Honeywell Technology Center. The RTARM system supports provision of integrated services for real-time distributed applications and offers services for end-to-end QoS negotiation, QoS adaptation, real-time application QoS monitoring and hierarchical QoS feedback adaptation. In this paper, we focus on the hierarchical architecture of RTARM, its flexibility, internal mechanisms and protocols that enable management of resources for integrated services. The architecture extensibility is emphasized with the description of several service managers, including an object wrapper build around the NetEx real-time network resource management system developed by the Texas A&M University. We use practical experiments with a distributed Automatic Target Recognition application and a synthetic pipeline application to illustrate the impact of RTARM on the application behavior and to evaluate the system's performance.

Key words: adaptive resource management, distributed real-time applications, integrated services, QoS negotiation and adaptation, hierarchical feedback adaptation

1. Introduction

Current distributed mission-critical environments employ heterogeneous resources that are shared by a host of diverse applications cooperating towards a common mission goal. These applications are generally a mix of hard-, soft- and non-real-time applications with different levels of criticality and have a variety of structures, ranging from periodic independent tasks, multimedia streams and parallel pipelines, to event-driven method-invocation communicating modules. The applications usually tolerate a range of Quality of Services (QoS) and are ready to trade off QoS in favor of the most critical functions they perform. The distributed systems must be able to evolve and adapt to the high variability in resource demands and criticality of the applications as well as to the changing availability of resources.

The current industry trend is to build distributed environments for mission-critical applications using "Common-Off-the-Shelf" (COTS) commercial hardware and software components. A middleware layer above the COTS components provides consistent management for the system resources, decreases complexity and reduces development costs.

This paper presents the Real Time Adaptive Resource Management system (RTARM), developed at the Honeywell Technology Center, that implements a general middleware architecture/framework for adaptive management for integrated services aimed to real-time mission-critical distributed applications.

The RTARM system has the following basic features [4]: (1) scalable *end-to-end criticality-based QoS contract negotiation* that allows distributed applications to share common resources while maximizing their utilization and execution quality; (2) *end-to-end QoS adaptation* that dynamically adjusts application resource utilization according to their availability while optimizing application QoS; (3) *integrated services* for CPU and network resources with end-to-end QoS guarantees; (4) real-time application *QoS monitoring* for integrated services and (5) *plug-and-play* architecture components for easy *extensibility* for new services.

The resource management architecture for RTARM uses an innovative approach that unifies heterogeneous resources and their management functions into a hierarchical uniform abstract service model [4]. The building block of the architecture is the Service Manager (SM). It encapsulates a set of services and their management functions and exports a common interface to clients and other service managers. This facilitates recursive hierarchies, in which heterogeneous services are integrated bottom-up. A higher-level service manager aggregates services provided by itself and its lower-level SMs and provides clients with a higher-level QoS representation.

In this paper, we focus on the architecture, protocols and implementation of an RTARM prototype that supports integrated services for real-time distributed applications. It runs as a middleware on a network of workstations and uses CORBA for portable communication. A major contribution of our work is the hierarchical feedback adaptation mechanism [1] that provides efficient dynamic QoS control for distributed data-flow applications. We illustrate the RTARM capabilities with a practical experiment with an Automatic Target Recognition (ATR) [8] distributed application and with a synthetic pipeline demonstration application.

Other efforts for building adaptive management systems for heterogeneous resources are GRMS [5,6], ARA [8,10], and QualMan [9]. GRMS is a precursor of RTARM. It introduced the uniform resource model and the atomic ripple scheduling protocol. Its hierarchical architecture reflects the application data flow and does not offer feedback adaptation. ARA considers a discrete set of runtime configurations for distributed applications and does feedback adaptation by resource reallocation. The ARA architecture is non-recursive and differs considerably from the uniform RTARM architecture by using proxies for specific service providers. QualMan is designed for multimedia applications and defines two basic resource management components, the resource scheduler and the QoS broker, that adhere to a uniform resource model without considering deeper recursive structures and QoS composition.

The rest of this paper is organized as follows. Section 2 describes the RTARM hierarchical architecture, system models and interfaces. Section 3 presents the architecture of a SM and describes the CPU, network and a higher-level SM. Section 4 continues with experiments involving an ATR application and synthetic pipeline applications that emphasize the RTARM capabilities. The paper concludes in Section 5 with a discussion and future plans.

2. The RTARM System Architecture

We have designed and implemented the RTARM system prototype as a middleware layer above the operating system and network resources. The middleware approach provides the benefit of flexibility and portability but the increased distance to the basic resources makes fine-grained control difficult. The RTARM servers, developed in C++, run as user-level processes on Windows NT workstations and export a CORBA (Orbix [7]) interface to clients and applications. The RTARM model differentiates between clients and applications. A *client* is any entity that issues a request for services and negotiates a QoS contract that defines the allocated services. An *application* consumes services reserved by a client on its behalf and continuously cooperates with the resource management system to achieve the best available QoS while maintaining its runtime parameters within the contracted region. The QoS contract may change during the application lifetime.

2.1 The Service Manager Hierarchy

The RTARM system employs a hierarchical resource management architecture that facilitates provision of integrated services over heterogeneous resources. The uniform resource model [4] defines a recursive structural entity called *Service Manager* (SM) that encapsulates a set of resources and their management mechanism. At the bottom of the hierarchy are SMs that provide management functions for basic resources, such as CPU or network resources, and directly control resource utilization by application components. Higher level services are assembled on top of lower-level services, giving rise to a service hierarchy.

Resources as well as negotiation requests are treated uniformly across the entire hierarchy. Higher-level service managers (HSM) may act as clients for lower-level SMs (LSM). The hierarchy allows dynamic configuration as new service managers can join the system at any time. A request for an integrated service sent to an HSM may require resources from lower-level service providers. The admission protocol builds a virtual spanning tree over the

SM hierarchy that remains valid for the entire application lifetime. The SM hierarchy forms a directed acyclic graph, with SM as nodes and edges represented by the "uses-services-from" relation.

Figure 1 illustrates a simple RTARM hierarchy with two LSMs, a CPU and a Network SM, at the bottom of the hierarchy. Two clients request services from the two HSMs while applications are consuming CPU and network resources. Section 3 describes the service managers in more detail.

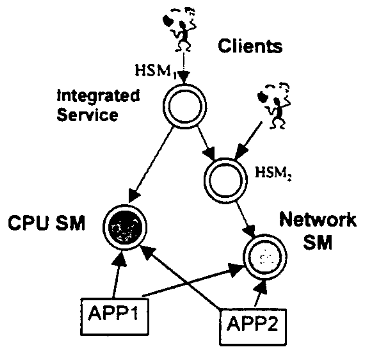


Figure 1. Sample RTARM hierarchy

There are several benefits from a hierarchical, recursive, resource management architecture. First, services with complex QoS representations are easier to implement on top of basic services. Complex distributed applications benefit from a richer representation of QoS. It simplifies the application design and facilitates consistent resource management for QoS-incompatible applications. Regardless of how complex the application architecture and QoS semantics are at the top of the SM hierarchy, at the bottom of the hierarchy everything translates to QoS requests for basic services (CPU and network in our implementation).

The hierarchical architecture of RTARM scales well with large distributed environments. Many SMs grouped in clusters may benefit from service localization and avoid communication bottlenecks. Sharing of LSMs between HSMs adds redundancy, fault tolerance and load balancing.

A potential drawback for deep SM hierarchies comes from the increased distance between the top-most-level SM and bottom layer in the hierarchy. This may cause high latency for time sensitive RTARM functions, such as feedback adaptation and application control.

Issues related to deadlock prevention and distributed SM synchronization have been studied for the GRMS project [5,6] and can be easily extended to the RTARM model.

2.2 RTARM System Models

QoS Model and Translation

The quality of the interaction of a mission-critical application with a dynamic environment directly reflects its performance. The wide magnitude of this interaction requires a range for the quality measures. RTARM supports a multidimensional QoS representation, each dimension specifying an acceptable range [Q_{min}, Q_{max}] of a quality parameter for the application. A set of range specifications, one per dimension, defines a QoS region. This QoS model facilitates resource negotiation and makes resource management more flexible.

In the RTARM recursive hierarchy, the QoS representation at a SM reflects the type of services provided by that SM. An HSM translates a QoS request for integrated services into individual QoS requests for services provided by itself and its lower-level SMs. When the SM receives replies from its LSMs, it reassembles the returned QoS into its own QoS representation in a process called *QoS reverse-translation.*

RTARM uses a unique implementation for QoS, which is independent of the addressed service. We define a QoS parameter as a set of *name-value* pairs, where the *value* part is a sequence of one or more scalar primitive data values (string, short, double, etc.) and the *name* indicates the specific QoS dimension, such as "rate", "workload", "latency", etc..

Adaptation Model

RTARM recognizes three situations when application QoS may be changed after admission [4]: (1a) *QoS shrinking/reduction* of lower criticality applications when a new application comes; (1b) *QoS expansion/improvement* when applications depart and release resources, and (2) *feedback adaptation*. While (1a) and (1b) imply contract changes and involve other applications, feedback adaptation does not change the contract but only varies the current operational point of the application within the contracted QoS region. Feedback adaptation is like closed loop control. It relies on monitoring of delivered QoS and uses the difference between delivered and desired QoS to adapt the application behavior.

2.3 RTARM Interfaces

Each SM implements and exports three interfaces: (1) `Negotiator` for admission control, collateral adaptation, QoS expansion and application control, such as suspend, resume and end; (2) `Service Manager` for SM hierarchy set up (register/deregister SM) and (3) `Monitor` for application monitoring and event propagation.

For admission control and adaptation RTARM uses a modified version of the GRMS *Ripple Scheduling* algorithm [5,6]. It consists of a transaction-based two-phase commit protocol applied recursively at each SM. The first phase executes a service availability test starting from the SM that received the admission request, down on the spanning tree that resulted from the QoS translation and request dispatch process. The available, reserved QoS propagates back to the initiator SM from the lowest SM layer, being reverse-translated along the way. In the second phase, the initiator SM assesses the success status of the reservation phase and the transaction is committed or aborted, implying service reservations along the spanning tree to be committed, or tobe cancelled, respectively. If not enough resources are available, a SM tries to adapt lower criticality applications at their minimum contracted QoS and use the released resources for the new application. Later, when resources become available, the SM expands the QoS for the most critical applications.

Sometimes in order to admit a new, more critical application, it is enough to squeeze the QoS of only a part of an existing distributed application. Then changes in the high-level QoS may require collateral adaptation of other components of the application that do not directly impact admission of the new application. For instance, for a multimedia stream application having frame rate as QoS parameter, if one processing stage is adapted to the minimum rate, than all other stages will run at the same low rate.

Next follows the list of calls from the RTARM CORBA interfaces:

The RTARM `Negotiator` interface for admission consists of the following set of calls:

- `boolean admit_app(in appId, in request, out admittedQoS)` -- admit new application; it embeds both phases of the ripple scheduling algorithm. Return admission status.
- `boolean test_reservation(in appId, in request, out admittedQoS, in candidateApps, out shrinkUsed, out adaptedAppsQoS, in hsmName)` -- phase I of admission protocol: try and reserve resources. Adapt candidateApps if necessary. Return admission status, QoS and adaptedApps.

- `boolean commit_reservation(in appId, in operationalQoS, in adaptedApps, out shrunkAppQoS)` and `boolean cancel_reservation(in appId, in adaptedApps)` -- phase II: commit/cancel reservation.

The RTARM `Negotiator` interface for collateral shrink:

- `boolean test_adapt(in newAppName, in appsToAdapt)` -- phase I for collateral adaptation: shrink QoS for the applications in appsToAdapt list, mark for newAppName.
- `boolean commit_adapt(in newAppName, in appsToAdapt, out adaptedAppsQoS)` and
- `boolean cancel_adapt(in newAppName, in appsToAdapt)` -- phase II: commit/cancel collateral adaptation for applications in appsToAdapt list.

The RTARM `Negotiator` interface for QoS Expansion:

- `boolean test_expansion(in appId, out availableQoS)` — phase I: try expansion for application appId and return status and `availableQoS`. If success, then services have been reserved.
- `boolean commit_expansion(in appId, in commitedQoS)` and
- `boolean cancel_expansion(in appId)` -- phase II: commit expansion to `committedQoS` or cancel.

The Negotiator interface for application control:

- `void end_app(in appId)` -- terminate application.
- `void suspend_app(in appId)` -- suspend application.
- `void resume_app(in appId)` -- resume execution.
- `void set_qos(in appId, in newqos)` -- change application QoS.
- `QoS get_qos(in appId)` -- get application QoS.

Service manager interface for the SM hierarchy setup:

- `boolean register_lsm(in name, in myParams, out hsmParams, out monitor)` -- register self as an LSM with the CORBA `name` at an HSM. Return HSM parameters and HSM `monitor`. The SM parameters include server name and a list of services it provides (cpu, network, pipeline,...).
- `boolean register_hsm(in name, in myParams, out lsmParams, in monitor)` -- register self as an HSM with the CORBA `name` at an LSM. Pass my parameters and my `monitor`. Return LSM parameters.
- `boolean deregister_sm(in smName)` -- remove SM with name `smName` from the list of SMs.

A SM cannot register twice to the same SM, but can be LSM and HSM for SMs in two distinct sets.

The Monitor interface for event communication and QoS reporting:

- `oneway void event(in appld, in originator, in event, in type)` -- send event to SM Monitor.

The next section presents the object architecture of the SM and details the implementation of a CPU, a Network and a Higher-level SM.

3. RTARM Service Managers

3.1 The Service Manager Architecture and Implementation

The unified resource model provides the benefits of a uniform internal architecture for all service managers (shown in Figure 2) and a common interface between them.

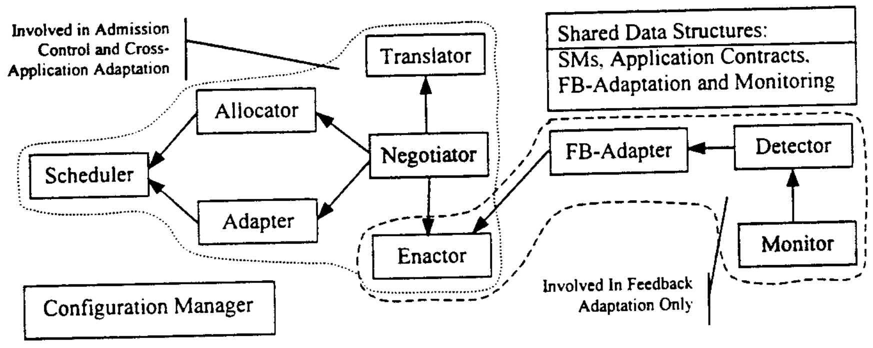

Involved in Admission Control and Cross-Application Adaptation
Translator
Shared Data Structures:
SMs, Application Contracts, FB-Adaptation and Monitoring
Allocator
Negotiator
FB-Adapter
Detector
Scheduler
Adapter
Enactor
Monitor
Configuration Manager
Involved In Feedback Adaptation Only

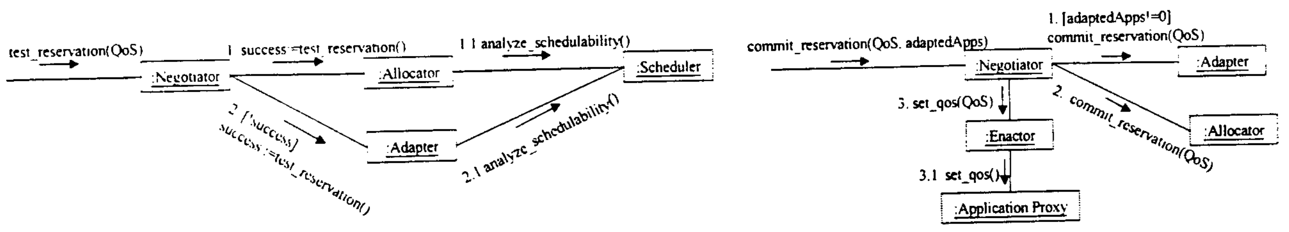
Figure 2. The internal object architecture of a service manager

The arrows in the figure indicate object service requests. The components in a SM are as follows:

- `Negotiator`: brokers contract admission, delegates responsibilities to other components and exports external RTARM CORBA interface.
- `Translator`: translates higher-layer integrated QoS into lower-layer QoS representation.
- `Allocator`: handles resource allocation/release when no adaptation is necessary.
- `Adapter`: handles resource allocation/release with adaptation and QoS expansion/contraction.
- `Scheduler`: determines whether allocation of resources and expansion of application QoS are feasible.
- `Enactor`: enforces changes in application QoS or status.
- `Monitor`: keeps an eye on applications in execution and passes status information and QoS usage to the `Detector`. Exports external RTARM CORBA interface.
- `Detector`: uses application runtime information (e.g. current QoS operational point) to detect significant changes in application operation (e.g. overload, underutilization, contract violation). Triggers `Feedback Adapter` actions.
- `Feedback Adapter`: decides corrective actions for applications when their runtime status changes significantly.

Additional data structures exist to hold information regarding application contracts, other service managers and available services.

As an illustration of the SM component interaction, Figure 3 shows object collaboration diagrams for two relevant interface calls for admission, `test_reservation()`and `commit_reservation()`, as they implement phases I and II of the admission protocol for a CPU SM.


test_reservation(QoS)
:Negotiator
1 success :=test_reservation()
:Allocator
1.1 analyze_schedulability()
:Scheduler
2 [!success] success :=test_reservation()
:Adapter
2.1 analyze_schedulability()
commit_reservation(QoS, adaptedApps)
:Negotiator
1. [adaptedApps!=0] commit_reservation(QoS)
:Adapter
2. commit_reservation(QoS)
:Allocator
3. set_qos(QoS)
:Enactor
3.1 set_qos()
:Application Proxy

a) `Negotiator::test_reservation()` b) `Negotiator::commit_reservation()`

Figure 3. Sample collaboration diagrams for the `Negotiator` admission interface

Applications implement a simple CORBA interface that allows SMs to change their QoS and status. LSMs keep proxies for the application CORBA server objects. All RTARM CORBA servers and applications are started in the shared, multi-client activation mode.

A SM component class has the same object interface regardless of the SM position in the hierarchy or the resources the SM controls. For instance, the `Adapter` object implements the same functions in all SMs, but in a way that depends actually on the scope of the SM. Not all components are required within a SM. For example, a `Translator` may exist only inside an HSM.

RTARM provides a common object oriented execution framework that allows users to dynamically load SM components from shared libraries during runtime configuration. A configuration manager uses a mechanism similar to a *Factory Method* [3] to instantiate SM components. It also passes configuration information extracted from a configuration file to the SM components during their initialization. For all SMs there is a single executable program that originally contains the empty SM framework and the configuration manager. By loading specialized components from shared libraries, the configuration manager practically starts different SMs. We use this technique when we initialize the CPU, Network and Higher-level SMs with components from specific Windows NT DLLs.

The flexibility of this plug-and-play feature permits implementation of a new SM by just replacing a set of components that realize a particular SM component interface, without rewriting the whole program. Writing a new SM component only requires the header file with the object interface, the executable program (common execution framework) and its corresponding library.

3.2 The CPU Service Manager

The CPU SM provides periodic applications access to a processor resource. Each computing node has a CPU SM, allowing concurrent applications to share a CPU. The application QoS is bi-dimensional: application execution rate (R) and iteration execution time (W) (Figure 4). The COP (Current Operational Point) represents the current values for the multidimensional QoS.

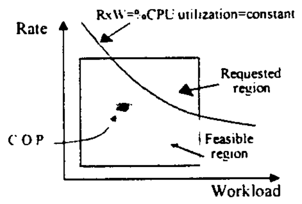


Figure 4. CPU SM QoS

Admission and Adaptation

The specific CPU scheduling policy is isolated within the Scheduler object and the Monitor keeps track of application CPU utilization. The invariant condition for admission and schedulability for n applications is $\Sigma_{i=1..n}R_iW_i < 100\%$ processor utilization. A more sophisticated CPU SM can be implemented at any time, by just using the plug-and-play feature, replacing the default `Scheduler` component with one specific to the scheduling discipline used.

The CPU SM service allocation unit for each periodic application is the fraction of CPU utilization (R x W). The CPU SM communicates this information to applications and assumes they are well behaved and keep their process utilization below the allocated limits. The SM scheduler only assigns application rates and does not control the underlying OS scheduler. This policy works fine on a larger time scale and for our experimental purposes. For real-time performance one solution is to implement a soft real-time CPU scheduling server above the OS scheduler [9].

Commercial operating systems with soft real-time capabilities, like Windows NT and Solaris, limit the scheduler granularity to 10-20ms.

The CPU SM implements the Ripple Scheduling admission protocol. Because it is at the bottom of the SM hierarchy and has no LSMs, it does not make any other recursive calls. Adaptation and collateral adaptation (Sections 2.2, 2.3) reduce the application rate to the minimum contracted value. QoS expansion increases the application contracted QoS (rate) to the best available value.

Feedback Adaptation

The CPU SM controls the task rate in real-time. It cannot change the workload, which is left exclusively under application control. Applications send their current QoS operational point as events to the CPU SM monitor at the end of each periodic iteration. At any moment, the QoS COP may vary so that $R \times W \leq L$, where L is the fraction of the contracted processor utilization. The CPU SM adjusts the COP as follows: (1) increase rate when workload decreases; (2) decrease rate on overload, when the workload pushes the COP outside the contracted region.

3.3 The Network Service Manager

We integrated the NetEx real-time network management system [2,11] from Texas A&M University into the RTARM system. NetEx runs as a middleware and provides connection-oriented real-time communication with guaranteed delay and bandwidth over COTS network infrastructure, such as ATM and switched 10/100 Mbps Ethernet. NetEx uses a tri-dimensional QoS: period, delay and message size and adds the connection source and destination network addresses to the connection contract. The NetEx resource management interface is, however, incompatible with the RTARM interfaces. It has different semantics and it does not export the two-phase commit protocol. We built an object-oriented *wrapper* [3] around NetEx that hides the incompatibilities and exports the RTARM interface to clients, applications and HSMs (Figure 5). The wrapper method can be applied to integrate any service provider in the RTARM architecture.

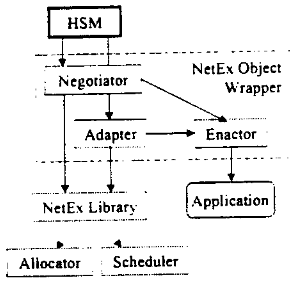


Figure 5. The NetEx Object Wrapper

The wrapper implements three SM components, `Negotiator`, `Adapter` and `Enactor`, that map the RTARM interface calls for admission, adaptation and expansion to the native NetEx API. NetEx does not provide feedback adaptation for connections, so the wrapper SM does not implement feedback adaptation either. It is important to note, however, that our HSM for integrated services for parallel pipeline applications implements hierarchical feedback adaptation. This is detailed in the next section 3.4.

3.4 The Higher-level Service Manager for Integrated Services

Within the RTARM service manager hierarchy, HSMs aggregate services from LSMs (CPU, Network or any other type of SM) and provide RTARM services to applications that need a more complex QoS representation. The unified resource model enables recursive deployment of HSMs. Our HSM implementation is generic and is able to support various types of distributed applications with arbitrary QoS representations that map to available LSM QoS. The only restriction is that the *Ripple Scheduling* admission and adaptation procedure and the hierarchical feedback adaptation must not contradict the applications semantics. The QoS Translator SM component inside an HSM is responsible for translating a QoS request into something the LSMs understand. Replacing the translator component with a different one (for a different QoS representation) produces a HSM capable of supporting different integrated services.

Admission and Adaptation

Figure 6 shows the pseudo-code for the recursive two-phase admission protocol that runs at the heart of each HSM:

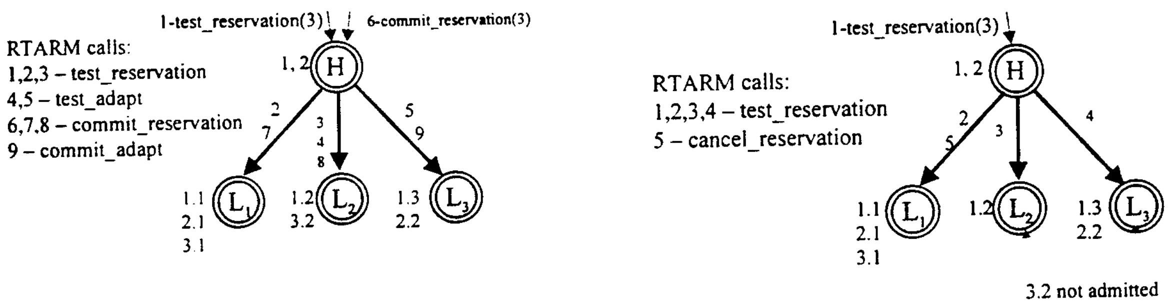
```
test_reservation(reqQos, avQos, candidates, adaptedApps) {
   translate reqQos into: LS - list of requested services from LSMs, and
                          LreqQos - corresponding QoS per service.
   for each service S from LS {
      for each LSM lsm that provides service S {
         success = lsm->test_reservation(LreqQoS[lsm], lsmAvQos[S],
                                         candidates that run on lsm, lsmAdaptedApps[S])
         if success then mark admitted service and continue with next service S from LS
      }
      if service S was not admitted then {
         cancel all previous successful admissions and
         return false
      }
   }
   // now all services from LS have been admitted
   reverse-translate and maximize the returned QoS from lsmAvQos into avQoS
   perform collateral adaptation if necessary
   return true
}

commit_reservation(committedQos, adaptedApps) {
   translate commitedQos into:
            Llsm - list of LSMs and
            LcommittedQos - committed QoS per service
   for each lsm from Llsm {
      lsm->commit_reservation(LcommitedQos[lsm], adaptedApps that run on lsm)
   }
   save committedQos into the application contract
}
```

Figure 6. Pseudo-code for the two-phase commit admission protocol

The `cancel_reservation()` call is similar to `commit_reservation()` and is omitted here.

Figure 7 illustrates examples of admission of a new application with id 3 at an HSM *H* that has 3 LSMs, L_1, L_2, L_3.

Applications 1 and 2 are already running at *H* and use services from L_1, L_2, L_3. For example, application 1 (denoted with 1 at *H*) runs also at L_1 (1.1), at L_2 (1.2) and L_3 (denoted 1.3). The new application 3 requires two services and maps to 3.1 and 3.2. In example a) both 3.1 and 3.2 are admitted at L_1 and L_2. Admission for 3.1 needs adaptation of application 1.1 on L_1. This triggers collateral adaptations for 1.2 as well as 1.3, as the entire application 1 must be adapted. Calls 4 and 5 (`test_adapt`) ask L_2 and L_3 to adapt collaterally application 1. During the execution of `commit_reservation` on *H* (call number 6), the collateral adaptation of 1 is committed on L_1 and L_2 with the two `commit_reservation` calls plus the extra `commit_adapt` call (9) to L_3. Example b) shows the call sequence when application 3 is accepted by L_1, but rejected both by L_2 and L_3. HSM H finally rejects 3 and returns **false** to the `test_reservation` call 1.



a) Successful admission of application 3 b) Failed admission of application 3

Figure 7. Examples of the admission protocol sequence executed at an HSM

We have implemented a Pipeline Service Manager (PSM), an HSM that aggregates services from lower-level SMs (CPU, Network, other HSMs) into a higher-level integrated representation suited for pipeline applications. Our PSM supports periodic independent tasks and periodic parallel pipeline applications, consisting of communicating stages in an arbitrary configuration, with a single source and a single sink node. We assume a sensor enters periodically data frames in the pipeline. Each frame is processed by a stage or a composite stage [1] (consisting of parallel strings of elementary stages) and then sent to the next stage. Such a pipeline application is depicted in Figure 8.

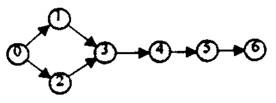


Figure 8. Parallel pipeline

For periodic pipeline applications, we use a QoS consisting of end-to-end message latency and rate for the final stage. The admission contract also contains execution time for each stage as well as the message size for each inter-stage connection. It is the job of the pipeline translator to decompose the integrated-service pipeline request into CPU and network admission requests. We assume all stages use the same range for rate. The pipeline QoS (end-to-end latency, frame rate plus state workloads and message sizes) translates into CPU QoS parameters for all stages and Network QoS for all network connections. The CPU QoS rate range is the same as that for the pipeline frame rate. The pipeline translator uses the same rate range and a fraction of the end-to-end pipeline latency to generate the Network QoS parameters.

Hierarchical Feedback Adaptation for Parallel Data-Flow Applications

We have implemented an innovative and efficient hierarchical feedback adaptation mechanism for parallel pipeline applications [1]. It performs feedback adaptation at two levels in the SM hierarchy. The pipeline end-to-end latency is controlled at the HSM level while the CPU SMs perform CPU feedback adaptation independent of the HSM.

The pipeline QoS parameter we consider critical and want to control is the end-to-end latency. As the pipeline evolves in time, rates of intermediate stages may change as a result of CPU SM feedback adaptation. In normal circumstances, the input sensor period is maintained at a value greater than the current period of any stage/substage of the parallel pipeline application, but it can get lower because of independent CPU feedback adaptation. When accumulation of queues between stages increases the end-to-end latency beyond a maximum threshold, the PSM sets the input sensor period at the maximum value from the pipeline contract. A finite state machine in the PSM maintains this maximal period for a fixed time, allowing the queues to empty. Then, the PSM sets again the input sensor period to the maximal current period of all stages, typically lower than maximum period from the contract. We have proved in [1] that the end-to-end latency decreases, and that after a finite number of frames the pipeline enters a region of stability where the end-to-end latency and the output frame rate are within the contracted region.

This method is simple and efficient, as the only parameter to be adjusted is the sensor input period, while the pipeline stages are controlled only by the corresponding CPU SM. This mechanism avoids costly communication and coordination between the HSM and all the CPU SMs. The information required for pipeline feedback adaptation is minimal: the end-to-end latency for the current frame and the maximal current period of all stages.

We present in the next section experiments with synthetic pipeline applications and an Automatic Target Recognition application to estimate the performances of the RTARM system.

4. Experiments and Performance Evaluations

To evaluate the RTARM system we designed two experiments. The first deals with synthetic pipeline applications and yields performance numbers for admission, adaptation and QoS expansion for the CPU, Network and Pipeline SMs. The second experiment tests feedback adaptation for parallel pipeline applications. The Forward Looking Infrared Automatic Target Recognition application provided an ideal testbed to prove the efficiency of our hierarchical feedback adaptation technique.

The runtime environment for these experiments consists of three 450MHz Dell Workstation-400 machines, running Windows NT, connected via a Fore ATM switch with OC-3c (155Mbps) links. Each machine hosts a CPU SM. Both the network SM that controls the inter-stage communication and the pipeline SM run on one of the three machines. We consider their own CPU resource consumption negligible. All inter-SM CORBA communication uses a secondary Fast Ethernet network, so the ATM lines remain 100% available. We used the NT performance counter for precise measurements.

4.1 Performance for Admission and Adaptation

For evaluating admission, adaptation and expansion for pipeline applications we devised two scenarios.

Scenario 1.

We tested admission of three-stage pipelines on a SM hierarchy with one HSM (P), one NSM (N) and two CPU SMs (C_1, C_2), as illustrated in Figure 9. The sequence of events is:

1. admit pipeline 1; no adaptation required.

2. admit pipeline 2 with higher criticality; stage 1.1 is adapted due to CPU constraints on SM C1; stages 1.2, 1.3 and network connections are adapted collaterally.
3. terminate pipeline 2; pipeline 1 is expanded back to its original QoS (all stages and the network connections).
4. try admission for pipeline 3 with lower criticality than 1; not enough CPU resources, admission is denied.
5. terminate pipeline 1.

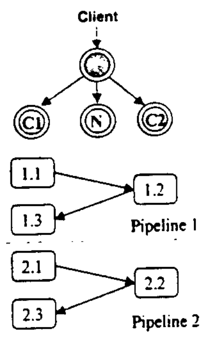


Figure 9. Scenario 1

Scenario 2.
Runs on the same environment as Scenario 1 and is similar, except the pipelines now have two stages and adaptation is caused only by network bandwidth constraints, not by CPU resource insufficiency.

Throughout the tests we measured the time to complete the RTARM interface calls for admission, adaptation and expansion for the CPU, Network and Pipeline SM. The measured time consists of the actual processing overhead and time to complete nested calls to: (1) application CORBA servers for the CPU SM; (2) the NetEx management subsystem and application CORBA servers for the Network SM (NetEx wrapper) and (3) LSMs for the Pipeline SM.

The performance measurements for the Pipeline SM are listed in Table 1, for the CPU SM in Table 2 and for the Network SM in Table 3. All values are expressed in milliseconds.

	w/o Adaptation		with Adaptation	
	Total time	Processing time	Total time	Processing time
test_reservation	99 159	17.972	118.344	18.899
commit_reservation	2239.02	6.366	2376.34	11.338
cancel_reservation	7.102	0.313		
test_expansion			212.751	4.508
commit_expansion			39.987	4.921
end_app	252.325	1 414	460.348	4.145

Table 1: Measurements for PSM

	w/o Adaptation		with Adaptation	
	with CORBA	w/o CORBA	with CORBA	w/o CORBA
test_reservation		0.447		0.707
commit_reservation	525.165	0 474	544 796	1.397
cancel_reservation		0.146		0.168
test_adapt				0.234
test_expansion				0 189
commit_expansion			3 132	0 112
end_app	4 619	0 846		

Table 2: Measurements for CPU SM

For the PSM the "Total Time" columns include the sequence of recursive RTARM CORBA calls to the LSMs and the algorithm processing overhead. Some calls may require adaptation of lower criticality applications, such as `test_reservation()` at step 2 in scenario 1; other calls, like the expansion operations, are 100% with adaptation. From Table 1 we notice that the reservation operations and `end_app()` require extra processing work if adaptation is involved. Also the processing time for `test_reservation()` is considerably larger than all other calls since it involves back-and-forth QoS translation and reverse-translation. But what stands out is the large total time consumed for `commit_reservation()` for a three stage pipeline application, approximately 2.3 seconds. This time includes the duration for `commit_reservation()` calls to the CPU SM that take more than 500ms for each pipeline stage (see Table 2). A CPU `commit_reservation()` call actually generates a `set_qos()` call with the committed application QoS to the application stage CORBA server. The stages are not up and running when admission happens. The Orbix daemon [7] starts the stage process and passes the CORBA server IIOP TCP port number and IP address to the CPU SM. Only after the stage is up and initialized it is able to respond to the`set_qos()` CORBA call from the CPU SM. The time to start a Windows GUI application (the pipeline stage) on Windows NT 4.0 is around half a second for our test configuration.

	w/o Adaptation				with Adaptation			
	Total time	Processing time	CORBA time	NETEX time	Total time	Processing time	CORBA time	NETEX time
test_reservation	22.475	3.147	0	19.328	48.414	3.901	0	44.513
commit_reservation	45.434	0.637	44.797	0	49.962	1.105	48.857	0
test_adapt					0.056	0.056	0	0
test_expansion					33.093	0.355	0	32.738
commit_expansion					0.697	0.697	0	0
end_app	10.08	0.289	0	9.791				

Table 3: Measurements for Network SM

Table 3 shows time measurements for the Nework SM These are more complex since the NetEx wrapper communicates through TCP/IP with the NetEx Host Traffic Manager [2,11] and stages through`set_qos()` CORBA calls (only during `commit_reservation()`). The communication latency overhead caused by NetEx is comparable to CORBA communication overhead, between 10 and 45ms.

We conclude that operation of the RTARM system is efficient, except the`commit_reservation()` call for CPU applications. This major delay can be completely avoided by pre-loading the applications before the client submits the pipeline contract to the HSM. The overall system performance may further improve by using a faster CORBA implementation that guarantees real-time operation deadlines.

4.2 Performance for Hierarchical Feedback Adaptation

The Automatic Target Recognition Experiment

We tested the RTARM feedback adaptation mechanism on a true mission-critical application. The ATR application, schematically shown in Figure 10, processes video frames captured by a camera and displays recognized targets on a display. Stage 0 (the sensor) generates frames that are passed through a series of filters and processing elements up to stage 6, which displays the original image and the identified targets. The frames are 8-bit, 360x360 pixels, monochrome images, and contain a variable number of targets (from 3 to 50), depending on the frame. Stages 4, 5 and 6 expose a variable workload, proportional to the number of targets, that without feedback adaptation would cause queue accumulations with negative effect on the end-to-end frame latency.

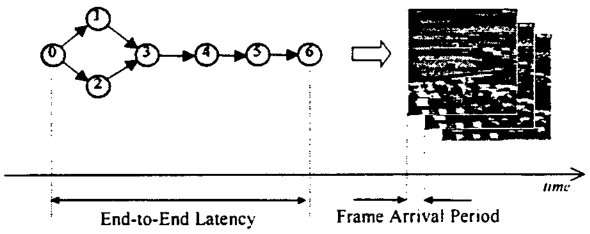


Figure 10. ATR pipeline application and QoS

Performance Metrics and Evaluation

The ATR pipeline contract requires an acceptable output frame period interval of [1,5] s, and a frame latency of 0.7-13 s. The seven ATR stages run at a variable workload between 0.02 and 1.5s and within the same period interval [1,5] s.

We first present timing measurements for the feedback adaptation at the CPU SM and PSM SM level (Figure 11). We measured the processing overhead of the feedback adaptation code (part 2 in Figure 11) and the time it takes the SM to react from the moment it receives the current QoS from the application until its adaptation command is enforced (part 2 + part 3).

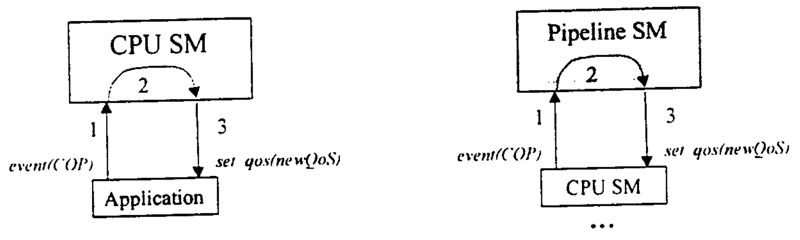


Figure 11. Feedback adaptation performance measurements.

The measured times are displayed in Table 4. For the CPU feedback adaptation, detection and enforcing the QoS adaptation takes around 4.4ms. Most of the time, 3.9ms, is spent in a `set_qos()` operation, a two-way normal, local CORBA call. The pipeline adaptation enforcement includes a `set_qos()` call to the CPU SM that controls the sensor (or first stage) that calls directly the first stage with a `set_qos()` call. This explains why enacting pipeline QoS adaptation takes almost double than for CPU SM QoS.

	Detection and decision processing (2)	**Decision Enactment (3)**	**Total Time (2+3)**
CPU SM	0.508 ms	3.914 ms	4.422 ms
Pipeline SM	0.859 ms	6.816 ms	7.675 ms

Table 4. Feedback adaptation performance results for CPU SM and PSM

Figure 12 displays CPU feedback adaptation for stage 4 in the ATR pipeline. The stage has variable workload that

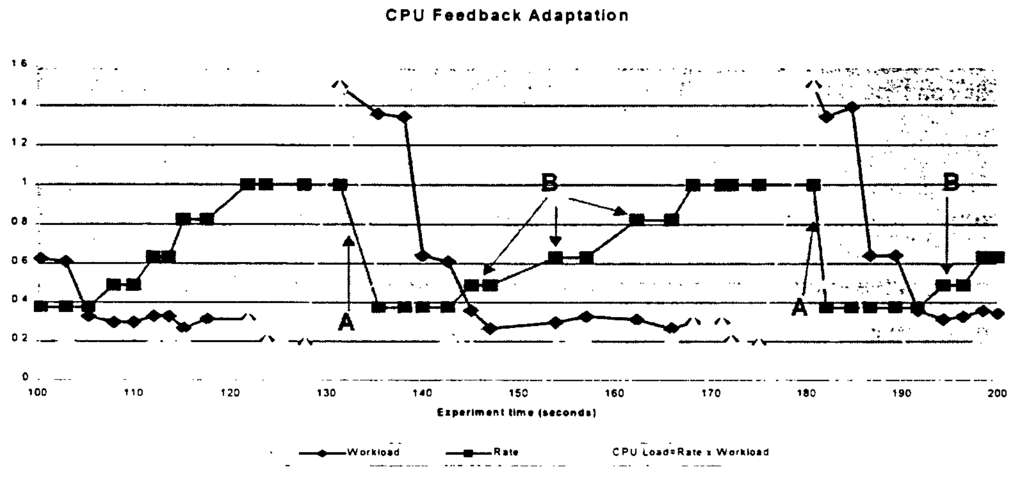


Figure 12. CPU SM feedback adaptation for a task with variable workload

triggers its CPU SM to change its rate. Points A indicate overload that triggers rate decrease and points B indicate chronic underutilization that determines rate increase.

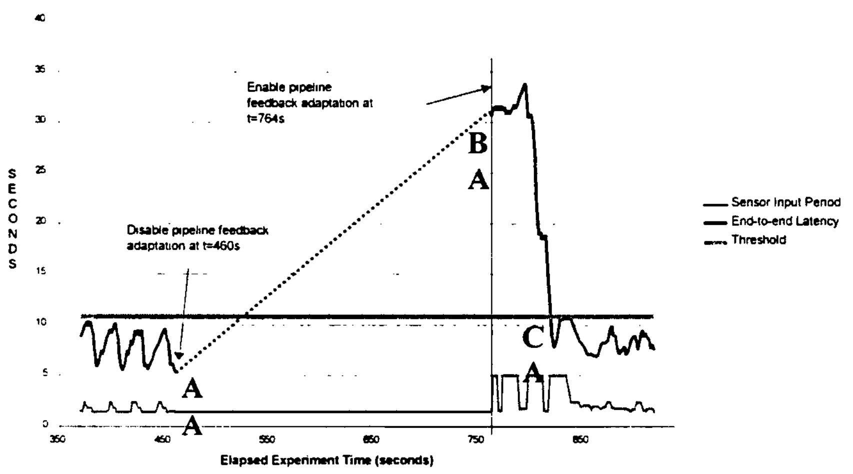


Figure 13. Latency Variation for ATR with and without pipeline feedback adaptation

While running the ATR application, the pipeline feedback adaptation mechanism makes sure the end-to-end latency and rate stay in the contracted range (Figure 13). In order to practically demonstrate its effectiveness, we disabled the pipeline feedback adaptation after some time while keeping the sensor input period at a sustained low value of 1.48s (0.67Hz). This caused accumulation of frames in stage queues that translated into an increasing end-to-end frame latency. While feedback adaptation was disabled we actually did not get latency measurements, so we drew a dotted line between points A and B. When the latency reached 30s, way above the contracted value, we re-enabled pipeline feedback adaptation. Immediately the PSM sensor increased the sensor input period up to 5s. The latency went rapidly down (B → C), below the threshold, after a brief spike caused by the inertia of the more than 23 frames already in transit through the pipeline.

Our hierarchical feedback adaptation algorithm proved to be effective and efficient. Detection, decision and enforcement take less than 8ms and involve only the CPU SMs for the sensor stage and the last stage that actually reports the latency and rate.

5. Conclusions

This paper presents the middleware architecture and implementation of the RTARM system. We have focused on the architectural elements that enable RTARM support for integrated services:

- the uniform service management recursive hierarchy and protocols
- the common architecture of a service manager that facilitates rapid OO prototyping, massive code reuse and features plug-and-play support for SM components.

Then we detailed the specific service managers (CPU, Network and Pipeline SM) that constitute the RTARM hierarchy. Finally, we presented experiments that illustrate the practical use of the RTARM system and its effectiveness for a real-world Automatic Target Recognition application. We demonstrated that our hierarchical feedback adaptation mechanism is able to efficiently control in real time the dynamic behavior of parallel pipeline distributed applications.

The clean and flexible architecture of a SM allowed us to integrate quickly a new service provider in the RTARM hierarchy. We built an object wrapper around the incompatible interface of the NetEx network management system that provided the same CORBA interface implemented by all RTARM service managers.

We plan to port RTARM to a real-time CORBA implementation, such as WUStL TAO [12] and to optimize its performance. We also intend to develop more sophisticated hierarchical feedback adaptation mechanisms with prediction features which would further decrease the system reaction time.

References

[1] Cardei, M., Cardei, I., Jha, R., Pavan, A., "Hierarchical Feedback Adaptation For Real-Time Sensor-based Distributed Applications", submitted to the 3rd IEEE International Symposium on Object-Oriented Real-time distributed Computing

[2] Devalla, B., Sahoo, A., Guan, Y., Li,C., Bettati, R., Zhao, W., "Adaptive Connection Admission Control for Mission Critical Real-Time Communication Networks", to appear in International Journal of Parallel and Distributed Systems and Networks, Special Issue On Network Architectures for End-to-end Quality-of-Service Support

[3] Gamma, E., Helm, R., Johnson, R., Vlissides, J., "Design Patterns. Elements of Reusable Object-Oriented Software", Addison-Wesley, 1994

[4] Huang, J., Jha, R., Heimerdinger, W., Muhammad, M., Lauzac, S., Kannikeswaran, B., Schwan, K., Zhao, W., Bettati, R.. "RT-ARM: A Real-Time Adaptive Resource Management System for Distributed Mission-Critical Applications", Proceedings of the IEEE Workshop on Middleware for Distributed Real-Time Systems and Services, December 1997

[5] Huang, J., Wang, Y., Cao, F., "On Developing Distributed Multimedia Services for QoS and Criticality Based Resource Negotiation and Adaptation", Journal of Real-Time Systems, May 1999

[6] Huang, J., Wang, Y., Vaidyanathan, N.R., Cao, F., "GRMS: A Global Resource Management System for Distributed QoS and Criticality Support", Proceedings of the 4th IEEE International Conference on Multimedia Computing and Systems, June 1997

[7] IONA Technologies. "The Orbix Programmer's Guide", 1997

[8] Jha, R., Muhammad, M., Yalamanchili, S., Schwan, K., Rosu, D., deCastro, C., "Adaptive Resource Allocation for Embedded Parallel Applications", Proceedings of the 3rd International Conference on High Performance Computing, December 1996

[9] Nahrstedt, K., Chu, H., Narayan., S., "QoS-aware Resource Management for Distributed Multimedia Applications", to appear in Journal on High-Speed Networking, Special Issue on Multimedia Networking

[10] Rosu, D., Schwan, K., Yalamanchili, S., "FARA – A Framework for Adaptive Resource Allocation in Complex Real-Time Systems", Proceedings of the 4th IEEE Real-Time Technology and Applications Symposium, June 1998

[11] Sahoo, A., Li, C., Devalla, B., Zhao, W., "Design and Implementation of NetEx: A Toolkit for Delay Guaranteed Communications", Proceedings of Milcom, December 1997

[12] Schmidt, D., Levine D., Mungee S., "The Design of the TAO Real-Time Object Request Broker", Computer Communications Special Issue on Building Quality of Service into Distributed Systems, Elsevier Science, 1998

Hierarchical Feedback Adaptation For Real Time Sensor-based Distributed Applications

Mihaela Cardei, Ionut Cardei, Rakesh Jha, Allalaghatta Pavan
Honeywell Technology Center
3660 Technology Drive
Minneapolis, MN 55418, USA
{mcardei, icardei, jha, pavan}@htc.honeywell.com

Abstract

This paper presents the Real Time Adaptive Resource Management (RTARM[1]) system, a middleware architecture for real-time adaptive resource management with support for integrated services, developed by the Honeywell Technology Center. This system is designed for distributed computing environments where mission-critical applications must be able to adapt to mission dependent variations in resource demands, as well as dynamic changes in resource availability. We describe the distributed hierarchical object-oriented architecture of RTARM, its flexibility and we focus on the issue of feedback adaptation in the RTARM system. Feedback adaptation is responsible for maintaining the application QoS parameters within the acceptable region and provides corrective actions triggered by significant events. The main contribution of this paper is a hierarchical feedback adaptation method that efficiently controls the dynamic QoS behavior of distributed data-flow applications, such as sensor-based data streams or mission-critical command and control applications. The method works independently at two levels in the RTARM hierarchy, at the distributed application level and at the CPU resource level. Our approach is simple and efficient. There is only one parameter that controls the application QoS at the distributed application level. Independently, the CPU service management level performs feedback adaptation to keep processor utilization within acceptable ranges. We present the analytical model for feedback adaptation applied to periodic distributed data-flow applications. We also describe experimental results for an Automatic Target Recognition distributed application and the impact of hierarchical feedback adaptation on the application behavior and its QoS parameters.

Key words: hierarchical feedback adaptation, distributed resource management, real-time applications, QoS negotiation and adaptation.

1. Introduction

Large, time-critical, distributed systems, such as defense computing environments, usually host a mix of application types that share common communication and processing resources. These applications exhibit a high degree of variability in performance requirements, criticality and demand fault tolerance and reliability. Building such a system using Common Of The Shelf (COTS) components is a challenge. In order to keep complexity under control, there is a definite need for an application Quality of Service-aware resource management system.

In recent years, there have been several efforts to build adaptive resource management systems for heterogeneous resources with real-time constraints [2,3,4,8,9,11]. This paper presents developments of the Real Time Adaptive Resource Management (RTARM) system [2], designed by the Honeywell Technology Center. The goal of the RTARM project is to develop a hierarchical real-time adaptive resource management system, implemented as middleware on COTS components and to apply it to mission-critical distributed applications.

The RTARM system defines a hierarchical resource management architecture that provides the following basic services [2]: (1) scalable *end-to-end criticality-based Quality of Service (QoS) contract negotiation* that allow distributed applications to share common resources while maximizing their utilization and execution quality; (2) *end-to-end QoS adaptation* that dynamically adjust application resource utilization according to their availability while optimizing application QoS; (3) *integrated services* for CPU and network resources with end-to-end QoS guarantees and (4) real-time application *QoS monitoring* for integrated services. An innovative feature of RTARM is the hierarchical resource management architecture that unifies heterogeneous resources and their management functions into a uniform abstract resource model. In this paper, we refer to services and resources interchangeably. The central piece of the architecture is the Service Manager, a recursive structural component. This encapsulates a set of services and their management functions. Because all service managers export the same common interface, it becomes easy to build layered hierarchies recursively, in which heterogeneous services are integrated bottom-up. This also helps rapid object-oriented prototyping and development.

The RTARM approach facilitates: (1) provision of integrated services and end-to-end adaptive QoS management. (2) easy extensibility to offer new service types, (3) design flexibility that provides affordable plug-and-play for architecture components as well as third-party service providers.

Many mission–critical distributed command and control applications, such as Automatic Target Recognition (ATR) [5], exhibit a degree of flexibility: they tolerate a range of QoS and resource usage above a minimum limit. Their performance depends on the allocated resources and they are ready to trade off some application service quality to save the critical services. For these applications, it is important to have a mechanism that regulates their dynamic behavior and protects them from contract violations. The main contribution of this paper is a new *hierarchical QoS-based real-time feedback adaptation* method for distributed periodic data-flow applications with parallel-pipeline structure. We have developed an analytical model that enables control of the end-to-end QoS behavior for the entire distributed application by adjusting the input rate in the pipeline. This model can be generally applied to any type of application with data-flow pipeline structure and a compatible QoS representation, such as multimedia streams and distributed command and control applications. We applied this model of feedback adaptation to our RTARM integrated service provider and experimented with a distributed ATR application.

Related work

Other adaptive real-time resource management systems are GRMS [3,4], ARA [9,10] and QualMan [8]. GRMS has a hierarchical structure that reflects the application data flow and does not offer feedback adaptation. The ARA framework [10] provides feedback adaptation for applications having a discrete set of acceptable configurations with specific resource needs. ARA accomplishes feedback adaptation by resource reallocation. [7] proposes a feedback adaptation method that adjusts the rate of data sent from a server to clients based on observation and prediction using a control-theoretical model. The system described in [6] uses digital control theory to determine the states of the adaptive system, which may activate control algorithms for adaptation. Another adaptive resource management system is QualMan [8], designed for distributed multimedia applications.

Our work differs from these approaches at the resource management architecture level, by supporting other application paradigms or by the way it accomplishes feedback adaptation.

The rest of this paper is organized as follows. In Section 2 we briefly describe the object oriented hierarchical architecture of the RTARM system, its interfaces and several service managers. Section 3 presents the feedback adaptation model and analysis for the periodic parallel-pipeline applications. Section 4 continues with the description of the hierarchical feedback adaptation in RTARM, the ATR experiment, performance metrics and evaluation and the impact of feedback adaptation on the ATR QoS. Section 5 concludes the paper and presents directions for future work.

2. The Real Time Adaptive Resource Management Architecture

We have implemented an RTARM prototype that supports periodic independent tasks and periodic parallel pipeline applications with real-time requirements. The RTARM system is built as a middleware layer above the operating system and network resources. RTARM allows service initiation requests (admission requests) from clients and views applications as service consumers. When a client requires a service from a service manager on behalf of an application, it negotiates a QoS contract that defines the allocated services. This contract may change later when the application is adapted.

The middleware approach provides the benefit of flexibility and portability but the increased distance to the real resources makes fine-grained control difficult. RTARM supports a multidimensional representation of QoS, defined by a set of parameters (e.g. rate, latency, jitter) specified as a range [QoS_{min}, QoS_{max}]. The RTARM system strives to allocate the best available services to applications with priority for ones that are more critical.

2.1 Hierarchical Adaptive Service Management for Integrated Services

The basic block of the RTARM recursive service manager hierarchy is the Service Manager (SM). It encapsulates a set of services and their management mechanism. At the bottom of the hierarchy are SMs that provide management functions for basic resources, such as CPU or network resources, and directly control resource utilization by application components. Higher level services may be built on top of lower-level services, giving rise to a service hierarchy. One use of a service hierarchy is to provide abstract or integrated resources for clients.

Figure 1 depicts a simple runtime configuration with two different Lower-level SMs (LSM), a CPU and a Network SM, at the bottom of the hierarchy, two applications and two clients accessing services from two Higher-level SMs (HSM).

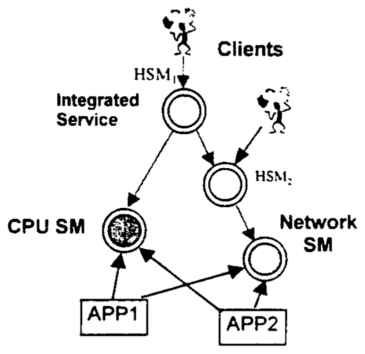


Figure 1. Sample RTARM hierarchy

Resources as well as negotiation requests are treated uniformly across the entire hierarchy. HSMs may act as clients for lower-level SMs that provide services to HSMs. The hierarchy allows dynamic configuration as new service managers can be added to the system anytime. Clients can directly access service providers at any point in the hierarchy, depending on their requirements. A request for an integrated service sent to an HSM may require resources from lower-level service providers. During the application admission procedure, a virtual spanning tree is built over the SM hierarchy that remains valid for the entire application lifetime.

2.2 Adaptation

RTARM recognizes three situations when application QoS may be changed after admission [2]: (1a) *QoS shrinking/reduction* of lower criticality applications when a new application comes; (1b) *QoS expansion/improvement* when applications depart and release resources, and (2) *feedback adaptation*. While (1a) and (1b) imply contract changes and involve other applications, feedback adaptation does not change the contract but only varies the current operational point of the application within the contracted QoS region. Feedback adaptation is triggered only by significant changes in application behavior, such as resource overload that results in a lowering of QoS operating point, resource underutilization that prompts RTARM to increase the application QoS operating point within the contracted QoS region and QoS contract violations that require corrective actions. Section 3 and 4 present feedback adaptation in detail for pipeline applications.

2.3 The Service Manager Object Architecture and Interface

The unified resource model approach for RTARM brings the benefits of a uniform architecture for all service managers and a common interface between them, implemented by us using CORBA. Figure 2 shows a simplified conceptual model of a service manager.

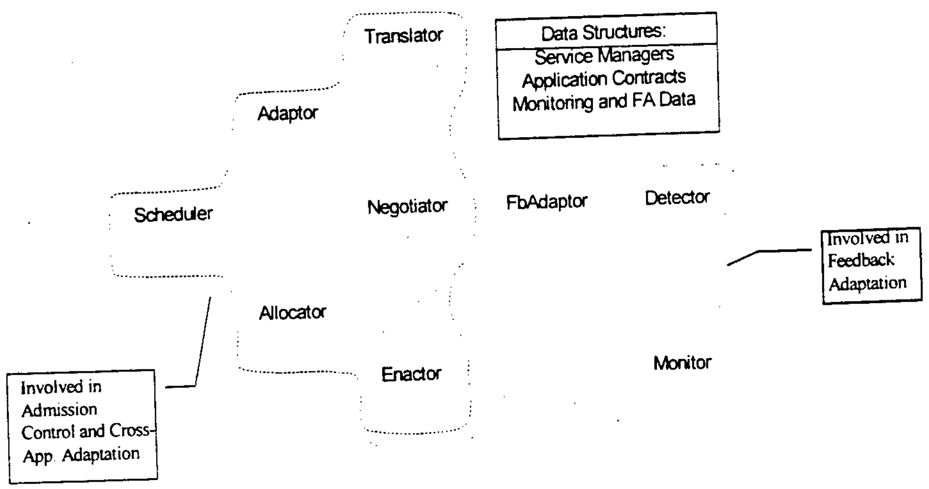


Figure 2. Service Manager simplified object model

A service manager is implemented as a user-level process with components implemented as communicating objects. The components in a SM are as follows:

- *Negotiator*: brokers contract admission, delegates responsibilities to other components and exports external RTARM CORBA interface.
- *Translator*: translates higher-layer integrated QoS into lower-layer QoS representation.
- *Allocator*: handles resource allocation/release when no adaptation is necessary.
- *Adapter*: handles resource allocation/release with adaptation and QoS expansion/contraction.
- *Scheduler*: determines whether allocation of resources and expansion of application QoS is feasible.
- *Enactor*: enforces changes in application QoS or status.
- *Monitor*: keeps an eye on applications in execution and passes status information and QoS usage to the Detector. Exports external RTARM CORBA interface.
- *Detector*: uses application runtime information (e.g. current QoS operational point) to detect significant changes in application operation (e.g. overload, underutilization, contract violation). Triggers Feedback Adapter actions.

- *Feedback Adapter*: decides corrective actions for applications when their runtime status changes significantly.

Additional data structures exist to hold information regarding application contracts, other service managers and available services.

The clear separation of functionality facilitates object reuse and flexibility. RTARM provides a common object oriented execution framework that allows users to dynamically load SM components (Scheduler, Adapter, ...) from shared libraries during runtime configuration.

Each SM implements and exports three interfaces. (1) Negotiator: admission control, collateral adaptation, QoS expansion and application control, such as suspend, resume and end; (2) Monitor: application monitoring and event propagation; (3) ServiceManager: service manager hierarchy set up, register/deregister SM.

RTARM uses a modified form of the *Ripple Scheduling* admission protocol from GRMS [3,4]. Admission, expansion and adaptation are transaction-based two-phase-commit protocols. User clients are shielded from the inter-SM admission protocol implementation details. A simple call*admit_app()* embeds the two phases and gives clients simple semantics for application admission.

2.4 Service Manager Instances

We currently have implemented for the RTARM project three service managers: CPU, Network and a higher-level, Pipeline SM. All follow the general SM internal architecture described in Section 2.3.

CPU Service Manager

The CPU SM provides periodic applications access to a processor resource. Each computing node has a CPU SM, allowing concurrent applications to share a CPU. The application QoS is bi-dimensional; the two parameters are application execution rate (R) and iteration execution time (W). The specific CPU scheduling policy is isolated within the Scheduler object and the Monitor keeps track of application CPU utilization. CPU feedback adaptation is presented in more detail in section 4.

Network Service Manager

We integrated the NetEx real-time network service manager [1,11] from Texas A&M University into the RTARM system. NetEx runs as a middleware and provides connection-oriented real-time communication with guaranteed delay and bandwidth over COTS network infrastructure, such as ATM and switched 10/100 Mbps Ethernet. NetEx uses a tri-dimensional QoS: period, delay and message size and adds the connection source and destination network addresses to the connection contract. The NetEx resource management interface is, however, incompatible with the RTARM interfaces. It has different semantics and it does not export the two-phase commit protocol. We built an object-oriented wrapper around NetEx that hides the incompatibilities and exports the RTARM interface to clients, applications and HSMs.

Pipeline Service Manager

The Pipeline Service Manager (PSM) is a higher-level SM that aggregates services from lower-level SMs (CPU, Network, other HSMs) into a higher-level integrated representation suited for pipeline applications. A PSM client can be a user or another HSM. The QoS Translator plays an essential role inside a PSM. It translates a request for integrated services into individual requests dispatched to LSMs.

Our PSM supports periodic independent tasks and periodic parallel pipeline applications, consisting of communicating stages in an arbitrary configuration, with a single source and a single sink node.

For periodic pipeline applications, we use a QoS consisting of end-to-end message latency and rate for the final stage. The admission contract also contains execution time for each stage as well as the message size for each connection. It is the job of the pipeline translator to decompose the integrated-service pipeline request into CPU and network admission requests.

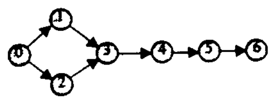


Figure 3. Parallel pipeline application

The *Ripple Scheduling* admission algorithm [3,4] fits well with our hierarchical recursive structure. A top-level admission request generates sequential recursive execution of the two-phase admission protocol at all intermediate layers in the resource allocation spanning tree. The PSM admits applications at the available QoS. The QoS expansion mechanism will provide later more resources to higher criticality applications and will boost their QoS operating point and the contracted QoS.

The PSM also provides hierarchical feedback adaptation (presented in section 4) that continuously monitors application QoS parameters and controls their resource utilization, taking corrective actions if necessary.

2.5 RTARM Flexibility and Plug-and-Play

All service managers have a similar internal architecture and each SM component has the same programming interface, regardless of the SM type and the resources it manages. This uniformity permits a common execution framework for all SMs. During SM initialization or at runtime, a configuration manager loads components from shared libraries. These can easily be replaced without recompiling the whole SM. For instance, we can get a Rate Monotonic Analysis-based CPU SM just by replacing the scheduler component. Our RTARM implementation has a single executable program and different SMs are instantiated just by loading RTARM SM components from different shared libraries. This increased flexibility allows quick prototyping and provides a plug-and-play feature for SM components developed by third parties.

2.6 Discussion

Work on the RTARM project is still in progress. The two-phase commit admission and adaptation protocols provide consistency and avoid the need for sophisticated synchronization between service managers. It also poses some scalability problems with deeper SM hierarchies. A deep SM hierarchy potentially would slow the reaction speed for feedback adaptation, as application monitoring information has to bubble through the hierarchy up to the HSM that got the admission request from the client.

The middleware approach itself brings extra performance penalties. Direct control over resources is difficult, and RTARM must rely on OS services or other middleware intermediate service managers. The increased flexibility, portability and the chance for rapid prototyping make the middleware implementation a reasonable compromise. The flexible SM architecture makes the implementation of the NetEx wrapper for the network SM easy.

Our current RTARM implementation runs on Windows NT machines and uses CORBA for inter-process communication. While Windows NT proved a stable development environment, we had our problems with its coarse-grained process scheduler and timer functionality. On the other hand, we found that CORBA fits well to the RTARM architecture. We plan to port RTARM in the near future to a real-time ORB and to optimize its performance.

3. Feedback Adaptation Model and Analysis for Pipeline Applications

This section presents a model for periodic pipeline applications and introduces an efficient and stable method for feedback adaptation. We consider the end-to-end latency as the most critical QoS parameter. The main result is that adjusting only the period for the input sensor can control the end-to-end latency of a pipeline application.

A pipeline application consists of stage tasks that process data sequentially. We assume a sensor enters periodically data frames in the pipeline. Each frame is processed by each stage in turn and then sent to the next stage. A clock-based pipeline assumes that each stage operation is synchronous and periodic. If a frame is available for processing at the beginning of a period, the stage will process and send it to the next stage(s) in the data flow. If no frame is available at the beginning of a period, the stage will block until the beginning of the next period, when it will repeat the same cycle.

Our model ignores the network communication overhead between two stages. This assumption would not affect the feedback adaptation for the Automatic Target Recognition experiment because of the large disparity between the stage period (1-5s) and communication latency (0.05s).

Our analysis assumes that the execution time and period of each stage are constant. These parameters may vary as the pipeline application evolves in time, and our analysis relates with a particular instance of time. It says that if starting with that moment the sensor input period is adjusted over some value, then, with the currently set parameters. the pipeline latency exhibit deterministic behavior. In this way, the analysis may be applied at any time instance for the corresponding parameters.

Section 3.1 presents our main results and an example for the clock-based simple pipeline and section 3.2 generalizes for clock-based pipeline with composite stages.

We have also analyzed the event-driven pipeline model, which may be useful for other types of applications. This model assumes the stages are aperiodic. They may start execution of a frame whenever it becomes available. The results obtained for this model are similar to those of the clock-based model: the sensor input period is the only factor the pipeline application needs to adjust to control the pipeline end-to-end latency. Due to the space limitation, we do not describe this model here.

3.1. Clock-Based Simple Pipeline

This section starts with the description of the clock-based pipeline application model, then presents the theoretical results for the control of the end-to-end pipeline latency and finalizes with an illustrative example.

The simple pipeline consists of individual applications (stages). Each stage receives a frame, processes it and then sends it to the next stage in the data flow.

Consider a pipeline with N+1 stages:

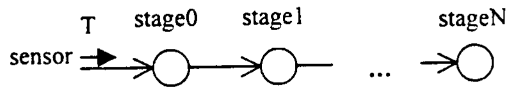


Figure 4. Linear, simple pipeline

Notations:

N+1 is total number of stages

T is the period at which the sensor pushes frames into the pipeline. It may change over time, but we assume it stays constant starting with the frame with which we develop the analysis.

C(i) is the execution (processing) time on stage i.

T(i) is the period of stage i, $T(i) \geq C(i)$.

W(i, n) is the waiting time for frame n, stage i. It represents the time the frame needs to wait before being processed by the stage i. It is greater than 0 if the stage i did not finish processing the previous frame.

$W(i, n) \geq 0$

$W(i, n) = \max [\, t_{out}(i, n-1) - t_{out}(i-1, n), 0 \,]$, where $t_{out}(i, n)$ is the time at which stage i produces output for frame n.

S(i, n) is the synchronization time. It is the time the frame n waits to synchronize with the beginning of the next period, for stage i. $0 \leq S(i, n) \leq T(i)$

$l(i, n)$ is the latency for frame n at stage i.

$l(i, n) = C(i) + W(i, n) + S(i, n)$

e(i, n) is the end-to-end latency up to and including the stage i, for frame n.

$e(i, n) = \Sigma_{j=0..i}\, l(j, n)$

L(n) is the end-to-end latency for the whole pipeline, for frame n.

$L(n) = e(N, n) = \Sigma_{j=0..N}\, l(j, n) = \Sigma_{j=0..N} C(j) + \Sigma_{j=0..N} W(j, n) + \Sigma_{j=0..N} S(j, n)$

Definition 1:

The pipeline is in the state S_k, where $0 \leq k \leq N$, for a frame x, if for all i = 0..k the relation (1) is true.

$e(i, x) \leq \Sigma_{j=0..i}(C(j) + T(j))$ (1)

Observation: If a pipeline is in the state S_k, then it is also in states $S_{k-1}, S_{k-2}, S_{k-3}, \dots, S_0$.

Definition 2:

We define the *stable region* for the end-to-end latency as the interval $[\ \Sigma_{j=0..N}\ C(j),\ \Sigma_{j=0..N}(\ C(j) + T(j)\ )\ ]$.

We say the pipeline is in the stable region if its end-to-end latency is within that interval.

If a pipeline is in the state S_N for frame x then it is in the stable region, because:

$$\Sigma_{j=0..N}\ C(j) \le L(x) \le \Sigma_{j=0..N}(\ C(j) + T(j)\ )$$

The left limit for L(x) is evident, because $L(x) = \Sigma_{j=0..N} C(j) + \Sigma_{j=0..N}\ W(j, x) + \Sigma_{j=0..N}\ S(j, x)$ and $\Sigma_{j=0..N}\ W(j, x) \ge 0$, $\Sigma_{j=0..N}\ S(j, x) \ge 0$.

From the application point of view it is important the pipeline latency be limited by an upper bound, because this guarantees it does not increase infinitely over time. The stable region of a pipeline corresponds to optimal pipeline behaviour, in the sense that its end-to-end frame latency is bounded. Next we present two theorems: the first one refers to the case when the pipeline is in the stable region and shows which sensor input periods maintain the pipeline there for the next frames. The second theorem handles the case when the pipeline is not in the stable region, and gives a solution which assures that the pipeline converges to the stable region after a finite number of frames.

Lemma 1 proves a useful relation, used in next two theorems' proofs.

Lemma 1:

If $W(i, n) > 0$ then the following relation is true:

$$e(i, n) = e(i, n-1) + T(i) - T \qquad (2)$$

Proof:

$W(i, n) > 0 \Rightarrow W(i, n) = t_{out}(i, n-1) - t_{out}(i-1, n)$

$t_{out}(i, n-1) > t_{out}(i-1, n) \Rightarrow S(i, n) = T(i) - C(i)$

$l(i, n) = C(i) + W(i, n) + S(i, n) = C(i) + W(i, n) + T(i) - C(i) = W(i, n) + T(i)$

$W(i, n) = t_{out}(i, n-1) - t_{sensor}(n-1) - t_{out}(i-1, n) + t_{sensor}(n-1)$

where $t_{sensor}(x)$ is the time instance when the sensor pushes the frame x

$W(i, n) = e(i, n-1) - (\ t_{out}(i-1, n) - t_{sensor}(n-1) - T\ ) - T = e(i, n-1) - e(i-1, n) - T$

$W(i, n) = e(i, n-1) - (\ e(i-1, n) + l(i, n)\ ) + l(i, n) - T \Rightarrow W(i, n) = e(i, n-1) - e(i, n) + l(i, n) - T$

Implies $e(i, n) = e(i, n-1) + T(i) - T$.

□

Theorem 1 refers to the case when pipeline is in the stable region. It proves that it is enough to maintain the sensor input period greater than the period of each stage in order to keep the pipeline in the stable region.

Theorem 1

If the pipeline is in the stable region for frame n-1 and the sensor input period $T \geq \max_{i=0..N} T(i)$, then the pipeline stays in the stable region for frame n.

Proof:

We show more generally, that if the pipeline is in the state S_k for a frame n-1, $0 \leq k \leq N$, and the input period $T \geq \max_{i=0..k} T(i)$, then the pipeline is in the state S_k for frame n.

We show by induction that $e(i, n) \leq \Sigma_{j=0..i}(C(j) + T(j)) \quad \forall\, i = 0..k$

Step1: for i=0. We show that $e(0, n) \leq C(0) + T(0)$).

We have one of the cases:

- $W(0, n) = 0$.

  $S(0,n) \leq T(0) \Rightarrow W(0, n) + S(0, n) + C(0) \leq T(0) + C(0) \Rightarrow e(0, n) \leq T(0) + C(0)$
- $W(0, n) > 0 \Rightarrow e(0,n) = e(0, n-1) + T(0) - T$ (use relation 2)

  $T \geq \max_{i=0..K} T(i) \Rightarrow T(0)-T \leq 0 \Rightarrow e(0, n) \leq e(0, n-1)$

  The pipeline is in state S_k for frame n-1 $\Rightarrow e(0, n-1) \leq T(0) + C(0) \Rightarrow e(0, n) \leq T(0) + C(0)$

Step2: suppose $e(i, n) \leq \Sigma_{j=0..i}(T(j) + C(j))$, for $i < k$. We show that $e(i+1, n) \leq \Sigma_{j=0..i+1}(T(j) + C(j))$

We have one of the cases:

- $W(i+1, n) = 0$

  $S(i+1, n) \leq T(i+1) \Rightarrow W(i+1, n) + S(i+1, n) + C(i+1) \leq T(i+1) + C(i+1)$

  We know that $e(i, n) \leq \Sigma_{j=0..i}(T(j) + C(j))$. Implies $e(i+1,n) \leq \Sigma_{j=0..i+1}(T(j) + C(j))$.
- $W(i+1,n) > 0 \Rightarrow e(i+1, n) = e(i+1, n-1) + T(i+1) - T$ (use relation 2)

  $T \geq \max_{j=0..K} T(j) \Rightarrow T(i+1) - T \leq 0$, implies $e(i+1, n) \leq e(i+1, n-1)$

  The pipeline is in state S_k for frame n-1 $\Rightarrow e(i+1, n-1) \leq \Sigma_{j=0..i+1}(T(j) + C(j))$

  Implies that $e(i+1, n) \leq \Sigma_{j=0..i-1}(T(j) + C(j))$.

  □

Theorem 2 refers to the case when the pipeline is not in the stable region. It provides a solution to the case when the pipeline latency is too high, and proves that it is enough to adjust the sensor input period in order to bring the pipeline end-to-end latency into the stable region, when the latency is superior limited.

Theorem 2

If the pipeline is NOT in the stable region for frame n-1 and starting with the frame n the sensor input period $T > \max_{i=0..N} T(i)$, then the pipeline converges into the stable region after a finite number of frames.

Proof:

Let us note the pipeline current state I, where $I \neq S_N$.

We show by induction that starting with frame n the pipeline behaves like:

$$\begin{array}{ccccccccc} I \rightarrow & S_0 \rightarrow & S_1 \rightarrow & S_2 \rightarrow & \ldots \rightarrow S_N \\ & m_0 & m_1 & m_2 & m_N \end{array}$$

where:

m_i is the number of frames needed by the pipeline in state S_{i-1} to converge in the state S_i, $0 \le i \le N$

$m_i \ge 0, \forall\, 0 \le i \le N$

Step1: show that $I \rightarrow S_0$ after a finite number of frames m_0

Suppose $I \ne S_0$ (otherwise we are done, with $m_0 = 0$).

We show that for each new arriving frame x, e(0,x) decreases compared with previous frame value, until it becomes less than T(0) + C(0), at which time the pipeline is in state S_0.

We have one of the cases:

- W(0, x) = 0

$e(0, x) = W(0, x) + S(0, x) + C(0) \le T(0) + C(0) \Rightarrow$ starting with this frame x the pipeline is in state S_0.

- W(0, x) > 0

e(0, x) = e(0, x-1) + T(0) - T (use relation 2)

$T > \max_{i=0..N} T(i) \Rightarrow T(0) - T < 0$, implies that $e(0, x) < e(0, x-1) \Rightarrow$ end-to-end latency up to the stage 0 decreases between frames x-1 and x.

The same process happens again over successive frames, until the pipeline gets in the state S_0. The number of frames after which the pipeline gets in state S_0 is :

$$m_0 = \left\lceil \frac{e(0) - (T(0) + C(0))}{T - T(0)} \right\rceil$$

where e(0) is the end to end latency up to stage 0, when pipeline is in state I.

Note: the greater the input period T, the smaller m_o, so the earlier the pipeline converges to stage S_0.

Step2: Suppose the pipeline is in the state S_i. We show that after a finite number of frames, m_{i+1}, the pipeline enters state S_{i-1}.

Suppose the pipeline is not in S_{i-1}(otherwise we are done with $m_{i+1} = 0$)

$\Rightarrow$ end to end latency up to the stage i+1 = $e(i+1) > \sum_{j=0..i+1}(T(j) + C(j))$

We show that for each new arriving frame x, e(i+1, x) decreases compared with previous frame value, until it becomes less than $\sum_{j=0..i+1}(T(j) + C(j))$, moment by which the pipeline is in state S_{i+1} .

We have one of the cases:

- W(i+1, x) = 0

$e(i+1, x) = e(i, x) + W(i+1, x) + S(i+1, x) + C(i+1) \le e(i, x) + T(i+1) + C(i+1)$

We know that $e(i, x) \le \sum_{j=0..i}(T(j) + C(j)) \Rightarrow e(i+1, x) \le \sum_{j=0..i+1}(T(j) + C(j))$

$\Rightarrow$ starting with this frame x the pipeline is in state S_{i+1} .

- $W(i+1, x) > 0$

  $e(i+1, x) = e(i+1, x-1) + T(i+1) - T$ (use relation 2)

  $T > \max_{j=0..N} T(j) \Rightarrow T(i+1) - T < 0 \Rightarrow e(i+1, x) < e(i+1, x-1)$

  $\Rightarrow$ end to end delay up to the stage i+1decreases between frames x-1 and x

The same behavior repeats over successive frames, until the pipeline gets in the state S_{i+1}. The number of frames after which the pipeline gets in state S_{i+1} is:

$$m_{i+1} = \left\lceil \frac{e(i+1) - \sum_{j=0}^{i+1}(T(j) + C(j))}{T - T(i+1)} \right\rceil$$

where e(i+1) is the pipeline end-to-end latency up to the stage i+1, at the instance the pipeline gets to state S_i.

Note: the greater the input period T, the smaller m_{i+1}, so the earlier the pipeline converges in stage S_{i+1}.

□

We have demonstrated that by increasing the sensor input period above the maximum period of all pipeline stages, the end-to-end latency converges to the stable region. The theoretical results presented before proved the stability of our pipeline control method.

Example

The next example illustrates how the pipeline end-to-end latency converges in time to the stable region when the input sensor period is increased above the maximum period of all stages.

Consider the following instance of a 9 stage pipeline:

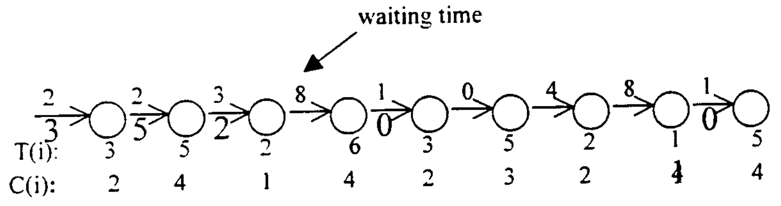


Figure 5. Example of linear pipeline

T(i), C(i), T, latency are represented in arbitrary time units. The end-to-end latency is 164, the stable region is [26, 68], and $\max_{i=0..8} T(i) = 11$.

In conformity with Theorem 2, if the sensor input period is greater than 11, the end-to-end latency converges to the

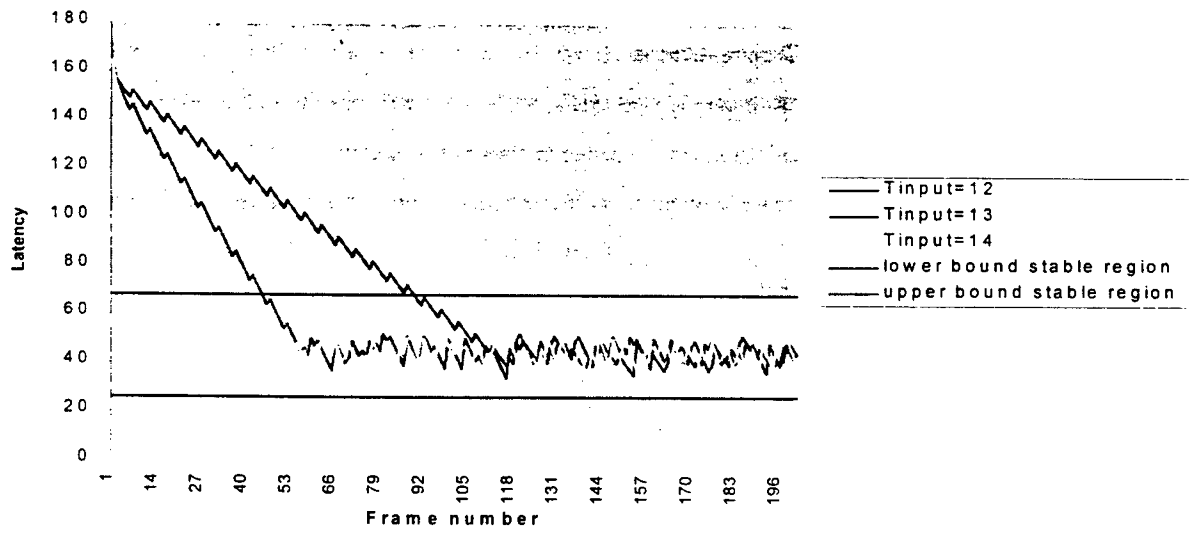


Figure 6. Latency variation depending on T_{input}

stable region. Figure 6 shows the pipeline behavior for T = 12, 13 and 14. We can observe that the greater the sensor input period T is, the earlier the pipeline enters the stable region. According to theorem 1, once the pipeline enters the stable region, it remains there as long as T ≥ 11.

3.2. Generalization for Clock-Based Pipeline With Composite Stages

This section generalizes the results achieved in the previous section. It presents the clock-based pipeline with composite stage model and the main results. Many distributed data-flow applications have a complex structure with branches and parallel substages. One example is the ATR application depicted in Figure 9. We model these architectures as a linear pipeline with simple and composite stages. Figure 7a. illustrates a simple stage and 7b. a composite stage.

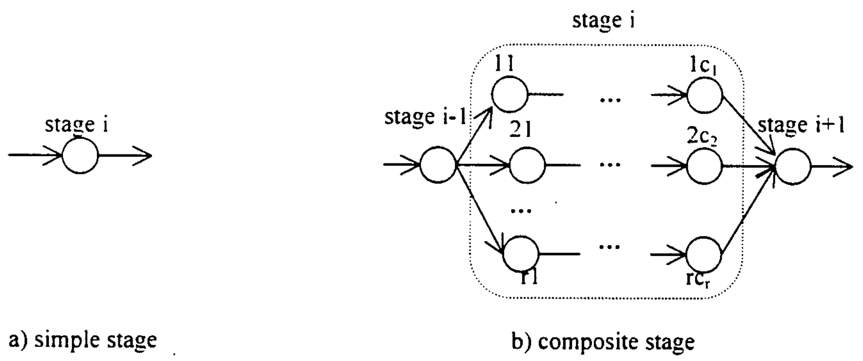


Figure 7. A simple and a composite pipeline stage

A simple stage represents a single, indivisible task that processes a frame. A composite stage i consists of substages arranged in parallel branches that process parts of a frame. A substage branch works like the simple linear pipeline presented in Section 3.1. When last substage of each branch finishes the processing, the frame is reassambled at stage i+1.

We proved that for this type of pipeline the results obtained previously for clock-based simple pipeline are valid: setting the input period greater than the maximum period of all stages/substages guarantees the pipeline convergence to the stable region after a finite number of frames. Once it enters the stable region, the pipeline remains there as long as the sensor input period is greater than the maximum period of all stages/substages. Due to space limitation we do not present here the formal proofs.

The next section describes how pipeline feedback adaptation works in RTARM applied to an ATR application. It gives also some measurements and performance evaluations.

4. RTARM Hierarchical Feedback Adaptation for Pipeline Applications

The top-most HSM that receives the admission request directly from the user client remains in control of the application QoS and its dynamics for its entire lifetime. That HSM is responsible for maintaining the distributed application's QoS within the contracted region and to improve it when possible using feedback adaptation. The resource management system must react quickly and adjust online the application parameters in case of allocated resource abuse or contract violation.

In RTARM we have designed and implemented an efficient hierarchical feedback adaptation mechanism and applied it to parallel pipeline applications and independent tasks, using the results developed in section 3. The RTARM hierarchy consists of a pipeline HSM, a network SM and several CPU SMs acting as LSMs. The network SM does not provide feedback adaptation. The reserved network resources must cover the entire range of application rate. According to our analysis, it is possible to control the end-to-end frame latency for the entire pipeline just by controlling the rate of the input sensor or first stage. This allows the CPU SMs to conduct local feedback adaptation for each individual pipeline stage in order to provide locally the best QoS within the contracted range. Thus, feedback adaptation for the entire pipeline and CPU stages is conducted independently.

CPU Service Manager Feedback Adaptation

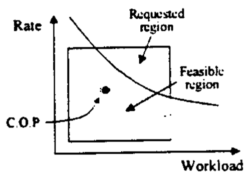


Figure 8. CPU SM QoS

CPU SMs run pipeline stages just like any regular periodic independent task. In fact CPU SMs have no idea these tasks are part of a higher level entity, and they perform all RTARM functions in the same way. As mentioned in section 2.4 the CPU SM QoS (Figure 8) consists of rate and iteration workload (execution time), both specified as intervals [min, max]. The CPU SM can directly control the application rate, but cannot touch the application workload. The CPU SM uses the product

CPU_utilization = *Rate* x *Workload* to asses schedulability. Applications send their actual QoS as events to CPU SM monitor at the end of each periodic iteration. The application is allocated a constant fraction L of the total processor time. At any time the current operational point (COP) may vary so that R x $W \leq L$. The CPU SM adjusts the current operational point:

- increase rate when workload decreases
- decrease rate on overload

Pipeline feedback adptation

The pipeline QoS parameter we consider critical and want to control is the end-to-end latency. As the pipeline evolves in time, rates of intermediate stages may change as a result of CPU SM feedback adaptation. In normal circumstances the input sensor period is maintained at a value greater than the period of any stage/substage of the parallel pipeline application, but it can get lower because of independent CPU feedback adaptation. When accumulation of queues between stages increases the end-to-end latency beyond a maximum threshold, the PSM sets the input sensor period at the maximum value from the pipeline contract. A finite state machine in the PSM maintains this maximal period for a fixed time, allowing the queues to empty. Then, the PSM sets again the input sensor rate to the maximal period of all stages. In this way, we know that the end-to-end latency decreases and after a finite number of frames the pipeline enters the stable region (section 3).

This method is simple and efficient, as the only parameter to be adjusted is the sensor input period, while the pipeline stages are controlled only by the corresponding CPU SM. This mechanism avoids costly communication and coordination between the HSM and all the CPU SMs. The information required for pipeline feedback adaptation is minimal: the end-to-end latency for the current frame and the maximal period of all stages.

Another option for pipeline feedback adaptation would have been to let the PSM directly adjust online the rate for each stage. In this case the PSM would have to keep track of the current workload and rate, and maybe queue lengths for all stages, implying extra communication, processing overhead and lower resource utilization for CPU service managers.

4.1. The Automatic Target Recognition Experiment

We tested the RTARM system and the feedback adaptation mechanism on a true mission-critical application. The ATR application, schematically shown in Figure 9, processes video frames captured by a camera and displays recognized targets on a display. Stage 0 (the sensor) generates frames that are passed through a series of filters and processing elements up to stage 6, which displays the original image and the identified targets. The frames are 8-bit monochrome images, 360x360 pixels and contain a variable number of targets (from 3

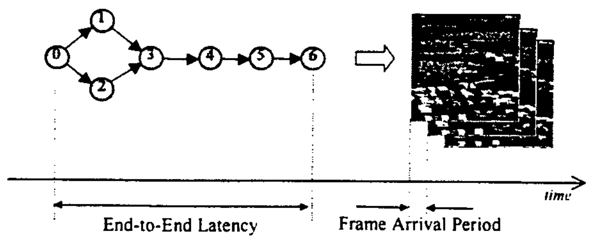


Figure 9: ATR pipeline application and QoS

to 50), depending on the frame. Stages 4, 5 and 6 expose variable workload, proportional to the number of targets, that without feedback adaptation would generate queue accumulations with negative effect to the end-to-end frame latency.

4.2. Performance Metrics and Evaluation

The runtime environment for the ATR experiment consists of three 450MHz NT Dell Workstation 400 machines, connected via a Fore ATM switch with OC-3c (155Mbps) links. Each machine hosts a CPU SM. Both the network SM and the pipeline SM run on one of those three machines, and we consider their own CPU resource consumption negligible. All inter-SM CORBA communication uses a secondary Fast Ethernet network, so the ATM lines remain 100% available. We used the NT performance counter for precise measurements.

The ATR pipeline contract requires an acceptable output frame period interval of [1,5] s, and a frame latency of 0.7-13 s. The seven ATR stages run at a variable workload between 0.02 and 1.5s and within the same period interval [1,5] s.

We first present timing measurements for the feedback adaptation at the CPU SM and PSM SM level (Figure 10). We measured the processing overhead of the feedback adaptation code (part 2 in Figure 10) and the time it takes the SM to react from the moment it receives the current QoS from the application until its adaptation command is enforced (part 2 + part 3).

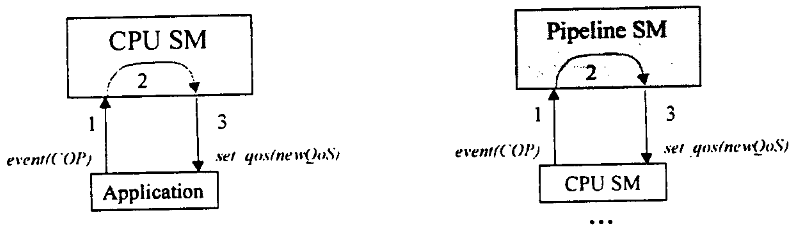


Figure 10. Feedback adaptation performance measurements.

The measured times are displayed in Table 1. For the CPU feedback adaptation, detection and enforcing the QoS adaptation takes around 4.4ms. Most of the time, 3.9ms, is spent in a *set_qos()* operation, a two-way normal CORBA call. The pipeline adaptation enforcement includes a *set_qos()* call to the CPU SM that controls the sensor (or first stage) that calls directly the application with a *set_qos()* call. This explains why enacting pipeline QoS adaptation takes almost double the time than that for CPU SM QoS.

	Detection and decision processing (2)	**Decision Enactment (3)**	**Total Time (2+3)**
CPU SM	0.508 ms	3.914 ms	4.422 ms
Pipeline SM	0.859 ms	6.816 ms	7.675 ms

Table 1. Feedback adaptation performance results for CPU SM and PSM

Figure 11 displays CPU feedback adaptation for stage 4 in the ATR pipeline. The stage has a period and variable workload and this causes the CPU SM to change the rate. Points A indicate overloads that trigger rate decrease and points B indicate chronic underutilization points that determine a rate increase.

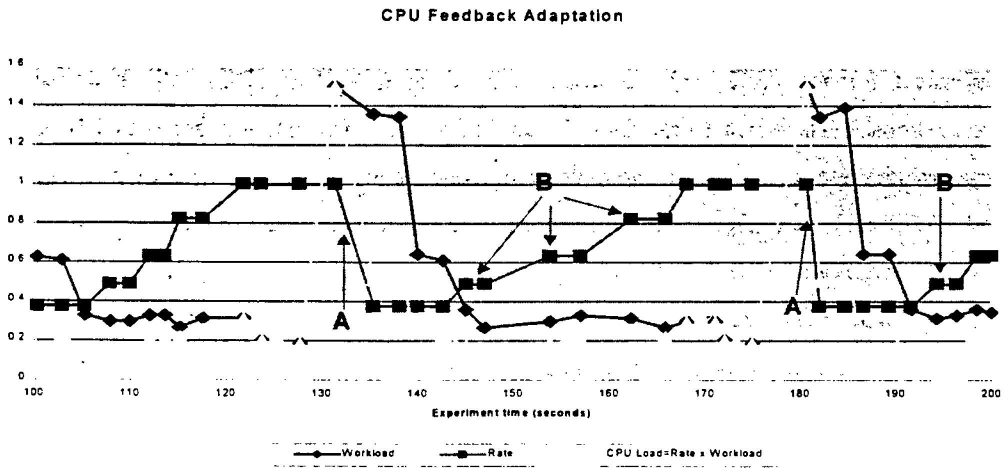


Figure 11. CPU SM feedback adaptation for a task with variable workload.

While running the ATR application (Figure 12), the pipeline feedback adaptation mechanism makes sure the end-to-end latency and rate stay in the contracted range. In order to practically demonstrate its effectiveness, we disabled the pipeline feedback adaptation after some time while keeping the sensor input period at a sustained low value of 1.48s (0.67Hz). This caused accumulation of frames in stage queues that translated into an increasing end-to-end frame latency. While feedback adaptation was disabled we actually did not get latency measurements, so we drew a dotted line between points A and B. When the latency reached 30s, way above the contracted value, we re-enabled pipeline feedback adaptation. Immediately the PSM sensor increased the sensor input period up at 5s. The latency went rapidly down (B → C), below the threshold, after a brief spike caused by the inertia of the more than 23 frames already in transit through the pipeline.

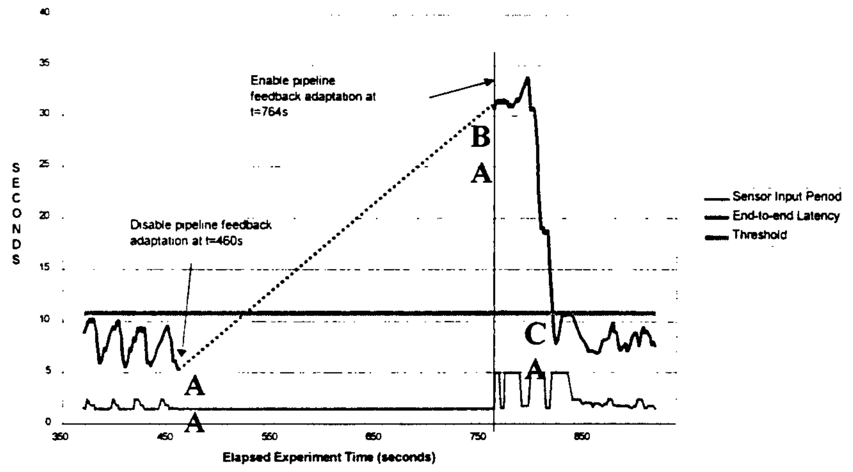


Figure 12: Latency Variation for ATR with and without pipeline feedback adaptation

Our hierarchical feedback adaptation algorithm proved to be effective and efficient. Detection, decision and enforcement take less than 8ms and involve only the CPU SMs for the sensor stage and the last stage that actually reports the latency and rate.

5. Conclusion

This paper presented briefly the Real-Time Adaptive Resource Management system, its architecture and flexibility. We developed a feedback adaptation mechanism for distributed data-flow applications based on an analytical model. We proved its correctness and stability and demonstrated its effectiveness by running an Automatic Target Recognition parallel pipeline application on a network of workstations managed by the RTARM system. Our innovative pipeline control method uses minimal information about the current state of the pipeline application and requires only one action to correct the end-to-end frame latency.

A direction for future work is to add prevention features to the current feedback adaptation method. Right now, it only takes corrective actions when the QoS falls below a threshold. Preventive actions would further decrease the overall pipeline reaction time. We also plan to study the feedback adaptation for parallel pipeline applications where several pipeline HSMs have exclusive control over parts (sub-pipelines) of the entire distributed application.

References

[1] Devalla, B., Sahoo, A., Guan, Y., Li,C., Bettati, R., Zhao, W., "Adaptive Connection Admission Control for Mission Critical Real-Time Communication Networks", to appear in International Journal of Parallel and Distributed Systems and Networks, Special Issue On Network Architectures for End-to-end Quality-of-Service Support

[2] Huang, J., Jha, R., Heimerdinger, W., Muhammad, M., Lauzac, S., Kannikeswaran, B., Schwan, K., Zhao, W., Bettati, R.. "RT-ARM: A real-Time Adaptive Resource Management System for Distributed Mission-Critical Applications", Proceedings of the IEEE Workshop on Middleware for Distributed Real-Time Systems and Services, December 1997

[3] Huang, J., Wang, Y., Cao, F., "On Developing Distributed Multimedia Services for QoS and Criticality Based Resource Negotiation and Adaptation", Journal of Real-Time Systems, May 1999

[4] Huang, J., Wang, Y., Vaidyanathan, N.R., Cao, F., "GRMS: A Global Resource Management System for Distributed QoS and Criticality Support", Proceedings of the 4th IEEE International Conference on Multimedia Computing and Systems, June 1997

[5] Jha, R., Muhammad, M., Yalamanchili, S., Schwan, K., Rosu, D., deCastro, C., "Adaptive Resource Allocation for Embedded Parallel Applications", Proceedings of the 3rd International Conference on High Performance Computing, December 1996

[6] Li, B., Nahrstedt, K., "A Control Theoretical Model for Quality of Service Adaptations", Proceedings of Sixth International Workshop on Quality of Service, 1998

[7] Li, B., Xu, D., Nahrstedt, K., "Optimal State Predication for Feedback-Based QoS Adaptation", Proceedings of Seventh IEEE International Workshop on Quality of Service, 1999

[8] Nahrstedt, K., Chu, H., Narayan., S., "QoS-aware Resource Management for Distributed Multimedia Applications", to appear in Journal on High-Speed Networking, Special Issue on Multimedia Networking

[9] Rosu, D., Schwan, K., Yalamanchili, S., "FARA – A Framework for Adaptive Resource Allocation in Complex Real-Time Systems", Proceedings of the 4th IEEE Real-Time Technology and Applications Symposium, June 1998

[10] Rosu, D., Schwan, K., Yalamanchili, S., Jha, R., "On Adaptive Resource Allocation for Complex Real-Time Applications", Proceedings of the IEEE Real-Time Systems Symposium, December 1997

[11] Sahoo, A., Li, C., Devalla, B., Zhao, W., "Design and Implementation of NetEx: A Toolkit for Delay Guaranteed Communications", Proceedings of Milcom, December 1997

*U.S. GOVERNMENT PRINTING OFFICE: 2000-510-079-81272

WAYNE A. BOSCO 5
AFRL/IFTB
525 BROOKS ROAD
ROME NY 13441-4505

HONEYWELL TECHNOLOGY CENTER 5
3660 TECHNOLOGY DRIVE
MINNEAPOLIS, MN 55418

AFRL/IFOIL 1 2
TECHNICAL LIBRARY
26 ELECTRONIC PKY
ROME NY 13441-4514

ATTENTION: DTIC-OCC 1 4
DEFENSE TECHNICAL INFO CENTER
8725 JOHN J. KINGMAN ROAD, STE 0944
FT. BELVOIR, VA 22060-6218

DEFENSE ADVANCED RESEARCH 1 7
PROJECTS AGENCY
3701 NORTH FAIRFAX DRIVE
ARLINGTON VA 22203-1714

ATTN: NAN PFRIMMER 1 12
IIT RESEARCH INSTITUTE
201 MILL ST.
ROME, NY 13440

AFIT ACADEMIC LIBRARY 1 17
AFIT/LDR, 2950 P.STREET
AREA B, BLDG 642
WRIGHT-PATTERSON AFB OH 45433-7765

AFRL/HESC-TDC 1 22
2698 G STREET, BLDG 190
WRIGHT-PATTERSON AFB OH 45433-7604

ATTN: SMDC IM PL 1 24
US ARMY SPACE & MISSILE DEF CMD
P.O. BOX 1500
HUNTSVILLE AL 35807-3801

COMMANDER, CODE 4TL0000 1 27
TECHNICAL LIBRARY, NAWC-WD
1 ADMINISTRATION CIRCLE
CHINA LAKE CA 93555-6100

CDR, US ARMY AVIATION & MISSILE CMD REDSTONE SCIENTIFIC INFORMATION CTR ATTN: AMSAM-RD-OB-R, (DOCUMENTS) REDSTONE ARSENAL AL 35898-5000	2	31
REPORT LIBRARY MS P364 LOS ALAMOS NATIONAL LABORATORY LOS ALAMOS NM 87545	1	33
ATTN: D'BORAH HART AVIATION BRANCH SVC 122.10 FOB10A, RM 931 800 INDEPENDENCE AVE, SW WASHINGTON DC 20591	1	35
AFIWC/MSY 102 HALL BLVD, STE 315 SAN ANTONIO TX 78243-7016	1	38
ATTN: KAROLA M. YOURISON SOFTWARE ENGINEERING INSTITUTE 4500 FIFTH AVENUE PITTSBURGH PA 15213	1	39
USAF/AIR FORCE RESEARCH LABORATORY AFRL/VSOSA(LIBRARY-BLDG 1103) 5 WRIGHT DRIVE HANSCOM AFB MA 01731-3004	1	44
ATTN: EILEEN LADUKE/D460 MITRE CORPORATION 202 BURLINGTON RD BEDFORD MA 01730	1	45
OUSD(P)/DTSA/DUTD ATTN: PATRICK G. SULLIVAN, JR. 400 ARMY NAVY DRIVE SUITE 300 ARLINGTON VA 22202	1	46
AFRL/IFT 525 BROOKS ROAD ROME, NY 13441-4505	1	1365
AFRL/IFTM 525 BROOKS ROAD ROME, NY 13441-4505	1	1366